TEMPORAL CONSTRAINT PROPAGATION FOR AIRLIFT PLANNING ANALYSIS

THESIS

Jeffery Dean Clay
Captain, USAF

AFIT/GCE/ENG/89D-1

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

TEMPORAL CONSTRAINT PROPAGATION FOR AIRLIFT PLANNING ANALYSIS

THESIS

Jeffery Dean Clay
Captain, USAF

AFIT/GCE/ENG/89D-1

AFIT/GCE/ENG/89D-1

TEMPORAL CONSTRAINT PROPAGATION FOR AIRLIFT PLANNING ANALYSIS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Jeffery Dean Clay, B.S.
Captain, USAF

December, 1989

Acknowledgments

I would like to thank everyone who helped me during my time at AFIT. This includes all of my friends, both from AFIT and otherwise who understood when things weren't going that great and kept telling me things would be better. Special thanks to my family who didn't seem to mind when I forgot birthdays, anniversaries, and so on. This thesis, although only my name is on the front, is a product of much time spent with my committee members, especially Capt Wellman from WRDC/TXI and LtCol Bisbee from AFIT/ENG. A very special thanks to Debbie for helping make part of my time spent at AFIT enjoyable.

Jeffery Dean Clay

Table of Contents

List of Figures

Abstract

Developing efficient airlift plans for large operations is difficult even for experienced planners. Time is often critical and days or hours may make the difference between success and failure. Airlift plans are developed and refined through a repetitive cycle to produce usable schedules. A planner selects resources for a plan, develops a trial schedule, and analyzes the schedule for weaknesses. This process is very time-consuming and a method is needed to analyze airlift plans and provide useful feedback early in the planning process. Temporal reasoning provides a general mechanism for such analysis. Different types of temporal constraints can be inserted into a network of airlift events to provide time bounds on execution of the complete plan. For this purpose we developed a general temporal constraint reasoner and a set of mechanisms for deriving temporal information from airlift requirements and partial schedule specifications. Physical limitations of the aircraft and operating facilities as well as the availability of cargo all provide constraints on when certain events may occur. These constraints may be the time required to fly from one location to another or the time spent waiting for an aircraft to be loaded. Comparing cargo requirements with airlift capacity over time provides additional constraints. The advantage of using a temporal constraint network as the underlying representation is its ability to accommodate various sources of information about time relationships between events in a plan. By asserting temporal information about specific events in an airlift plan, the planner can assess the impact of high-level planning decisions.

TEMPORAL CONSTRAINT PROPAGATION FOR AIRLIFT PLANNING ANALYSIS

I. INTRODUCTION

Background

Even the most experienced airlift planners find it difficult to develop an efficient plan for large operations. In a wartime environment, time is critical and days or even hours may determine the difference between success and failure. Developing an effective wartime airlift plan may require several weeks or more. The sheer complexity of the schedule and the number of choices available to the planner contribute significantly to the time required to produce an efficient plan. It simply is not possible to evaluate all possibilities for a large operation in a reasonable amount of time. As a result, airlift planning follows a hierarchical process. General plans are developed from scratch and then refined to produce a final schedule. A seemingly insignificant choice, made early in the planning process, may make a significant difference in the operational effectiveness of a plan.

Military Airlift Command (MAC) uses a program called MACPLAN to aid in developing deliberate airlift plans (2). MACPLAN is a planning tool developed for MAC by the MITRE Corporation. MACPLAN provides the human planner with an automated, menu-driven program for selecting aircraft and operating units to satisfy a given set of requirements. The planner makes the decisions and saves them using MACPLAN. Plans developed using MACPLAN are easy to change because the plan only needs to be modified in MACPLAN instead of being redeveloped. MACPLAN does not provide a detailed schedule, but instead provides a list of resources and requirements, called a planset. A planset consists of a list of requirements and a

list of aircraft and operating units with the number and type of aircraft specified for each day. A planset may be analyzed using MACPLAN to check the capacity of the chosen aircraft and to make sure the specified routes are within the flying range of the aircraft selected. Once a planset is generated, it is passed to FLOGEN (FLOw GENerator) for development of a specific timeline and final analysis by forward simulation. FLOGEN takes several hours to develop a schedule from a given planset. One poor choice in the planset may cause the schedule to be infeasible and require detailed manual analysis followed by another run on FLOGEN. This process may be repeated several times before an acceptable plan is found. For this reason, it is desirable that the planset given to FLOGEN be as close to a finished plan as possible.

Problem

The goal of this thesis is timely evaluation of a given planset to determine its feasibility and efficiency without using FLOGEN. The current analysis capability of MACPLAN had to be improved to accomplish this. MACPLAN provides the capability to evaluate such factors as total aircraft load capacity for each day, aircraft range, and the capacity of each ground station. MACPLAN's analysis provides a lower bound on the daily backlog of cargo which cannot be transported on the required dates. This backlog is based on a gross estimate of the airlift capacity provided for each day minus the requirements for that day. An analysis considering each individual requirement's source and destination would provide a more realistic backlog estimate. For example, if two requirements were listed on one day as going to different places and only one aircraft were scheduled for that day, MACPLAN would not show a backlog if both requirements would fit on the plane. However, it is obvious that one plane cannot be in two different places at the same time. Their relative locations would dictate the feasibility of satisfying both requirements in one day.

Currently, a planset is passed through FLOGEN to generate a schedule with the necessary detail for a useful analysis. By using a temporal reasoning system, the time constraints inherent in the requirements and physical limitations of the aircraft and ground stations can be asserted into a temporal network. This network provides more accurate measurements of the time required to transport each requirement than is provided by MACPLAN because of the additional detail considered in the temporal network. A temporal reasoning system is a good candidate for solving this type of problem because it integrates all constraints into a single network regardless of the source of the constraint.

Objective

An efficient method is needed to evaluate a proposed airlift planset for feasibility and efficiency before refining it to a completely specified schedule. A temporal reasoning system may provide useful information for analyzing general airlift plans without simulation by examining the times required to transport each requirement with the aircraft obligated in a planset. The times can then be translated into a temporal network of time-points and durations between the time-points. This allows the planner to analyze the planset and fix problem areas before performing a time-consuming FLOGEN schedule generation.

Analysis of an airlift plan early in the process is the objective of this thesis. A temporal reasoning system is proposed to analyze a high-level plan and to provide useful information about possible shortcomings of the plan. Additional information can be found using a more careful analysis of cargo requirements versus airlift capacity. These two techniques used together provide a useful analysis of airlift plans early in the planning process, thereby saving valuable time.

Scope

A general-purpose temporal reasoning system was developed as well as an interface to extract MACPLAN information and represent it in the temporal network. The capability to provide a detailed analysis of the airlift support versus the daily requirements was also developed. These two systems together provide more detailed information about a planset than MACPLAN provides. The temporal network provides an optimistic measurement of the time required to complete the plan while the backlog analysis provides an optimistic measurement of how much cargo can be moved on each day. This thesis is directed at providing an analysis of a developed planset which should yield a better estimate of the actual time required to execute the plan than does the estimate given by MACPLAN.

The program is written in Common LISP for the Symbolics computer. MACPLAN runs on the Symbolics and the temporal reasoning system was developed on the same machine in order to read MACPLAN data. Using Common LISP allows re-use of any useful code used in MACPLAN as well as exportability of code to other machines supporting Common LISP.

Approach/Methodology

Development of the enhanced analysis facility consisted of the following steps:

- Develop and test a temporal reasoning system.
- Develop a detailed analysis of cargo requirements versus airlift capacity.
- Interface the temporal reasoning and cargo analysis to MACPLAN.
- Evaluate the results.

The temporal reasoning system maintains information about the relationships between certain instances of events, or time-points. A detailed analysis of cargo requirements versus airlift capacity determines how much cargo can be moved with

available aircraft. The interface module reads a planset and asserts constraints about the relationships of time-points in the plan into the temporal network.

The temporal reasoning system is based on the temporal constraint representation scheme discussed by Dechter et al. (2). Time-points and durations are represented as nodes and links in the network. Time-points represent specific events such as take-offs, landings, or the onloading of cargo. Durations represent a bound on the time between two time-points. A typical assertion may be that an airplane may land between three and four hours after it takes off. These assertions are based on the constraints in the database such as aircraft type, ground station capabilities, cargo tonnage, the earliest available date for cargo, or the range of a specific aircraft.

The cargo analysis estimates how much of the daily cargo requirements can be transported using the available aircraft. Each requirement is mapped onto aircraft until either all cargo is moved or no aircraft are left. If an aircraft is not loaded to full capacity, part of the available airlift capability is not used. This reflects the real life constraint of using aircraft to transport needed cargo even when the aircraft are not full.

The interface module reads the MACPLAN database and extracts information such as aircraft type, onload station, offload station, and earliest available date. The system then generates time-points for the significant events of each requirement and asserts durations between these points based on the constraints in the data base. The system reads each requirement and maps the aircraft allotted in the planset to the cargo listed until either all of the cargo is moved or there are no more aircraft left. After all assertions are made into the temporal network, the system may be queried to find bounds on the predicted execution time for the system. Queries may also be made for the relative execution times of any two time-points in the network. For example, suppose a planner wanted to know the earliest time that a specific requirement could be delivered to its destination. A simple query provides a window of times between which the cargo may be delivered. The first time would

be the most optimistic and the second would be the worst possible case given all of the constraints posted. The worst possible case is usually infinity since the system constrains the event only with a lower bound and not an upper bound.

The predicted execution times generated by the temporal reasoner are expected to be greater than those predicted using MACPLAN. This is because MACPLAN uses less detail about the actual schedule in predicting execution times than is considered by the temporal reasoning system. MACPLAN and the temporal reasoning system work from the same database and both are estimators of partially specified schedules.

Materials and Equipment

A Symbolics 3600 computer using Common LISP was used to develop all code for this thesis. Version 18.0 of the MACPLAN program was obtained through WRDC/TXI as well as the plansets used to compare results.

Overview of the Thesis

This thesis describes a preliminary deliberate airlift plan analysis tool. Chapter 2 provides some detailed background on how current deliberate airlift planning is accomplished with MACPLAN. Chapter 3 reviews current temporal constraint representations and propagation techniques. Chapter 4 details the design and operation of the developed deliberate airlift analysis system. Both the temporal reasoning system and the capacity analyzing system are described. Chapter 5 discusses the problems encountered in developing the system and the results obtained. Chapter 6 lists proposed enhancements to the system and recommendations for further research.

II. THE MACPLAN FRAMEWORK FOR AIRLIFT SCHEDULING

Introduction

Military Airlift Command (MAC) is responsible for the development of airlift plans to support both wartime and peacetime requirements of the unified and specified commands as directed by the Joint Operations Planning System (JOPS). JOPS is the Department of Defense directed, Joint Chiefs of Staff specified system used in planning global and regional joint military operations, except the Single Integrated Operation Plans (SIOP). There are two types of planning performed at MAC: Crisis Action System (CAS) planning, and deliberate planning. Planning performed during peacetime is called deliberate planning, while CAS planning occurs during contingency and crisis situations in support of other unified and specified commands.

This thesis deals only with deliberate planning. The purpose of deliberate planning at MAC is to identify the total movement requirements, to describe them in logistic terms, to simulate the strategic deployment, and to produce a transportation-feasible Operation Plan (OPLAN) (1). An OPLAN is any plan, except the SIOP, for the conduct of a single military operation or series of connected operations prepared by the commander of a unified or specified command in response to a requirement established by the Joint Chiefs of Staff.

MACPLAN Operation

Deliberate airlift planning is currently performed by experienced planners at HQ MAC using MACPLAN and FLOGEN. These two programs together allow planners to take a set of requirements, identify a force package of particular types of aircraft, and generate a schedule to move the requirements with the specified aircraft. MACPLAN aids the planner in assembling the force package and FLOGEN generates

a schedule with the output of MACPLAN. If problems are found after analyzing the FLOGEN output, another cycle through the process is required to develop an acceptable plan. This process may take several days to develop a plan that is efficient and satisfies all requirements.

MACPLAN provides an automated tool for airlift planners to use in developing a force package to satisfy a set of requirements (4). MACPLAN's automation of this process provides an easier way to select and record these choices than direct encoding in FLOGEN format. Analysis capabilities in MACPLAN include range checking of the routes which must be flown, gross capacity checking for the aircraft selected, and aircraft compatibility with the ground stations selected. These features are limited in the amount of detail which is examined. The following sections describe MACPLAN's database and operating concepts.

MACPLAN operates using predefined LISP objects. These objects are the MACPLAN representation of the MAC world and contain information used by the planner in making decisions. These objects are based on actual assets of MAC and represent aircraft, airbases, and operational units. MACPLAN allows the planner to load in previously recorded requirements and assemble a set of aircraft to move the requirements. After selecting the aircraft, the planner chooses the routes to be flown by the aircraft from their home bases to pick up the cargo, and then throughout the journey to deliver the cargo. MACPLAN saves this as a planset containing the force package, the paths to be flown, and information about the ground stations to be visited.

The force package includes data about the aircraft to be used and the units that operate the aircraft. The data includes numbers of aircraft for each day (staging), configuration of each type of aircraft and other information about the aircraft. A different force package object is created for each type of aircraft selected by the planner. The information for the aircraft is itself an aircraft object contained in the force package object. The aircraft object contains such information as aircraft name,

```
($F (TYPE C141B) (WAA-TYPE C141) (FOUR-CHAR-MDS C141)
(UTE-RATE 10) (CREW-DAY "1600") (AUGMENTED-CREW-DAY "2400")
(CREW-REST-TIME "1200") (CREW-ALERT-TIME "0315") (TAS 425)
(TAKEOFF-FACTOR "0020") (MAX-RANGE "1110")
(CRITICAL-RANGE "0930") (COST-PER-HOUR 2520)
(ENGINE-RUNNING-OFFLOAD "0000") (ONLOAD-TIME "0215")
(OFFLOAD-TIME "0215") (ENROUTE-TIME "0215")
(AIR-REFUEL-TIME "0000") (AIR-DROP-TIME "0000")
(FUEL-1 15000) (FUEL-2 12500) (FUEL-3 12000)
(FUEL-CAPACITY 414000) (BULK-CAPACITY 24.0)
(OVERSIZE-CAPACITY 24.0) (OUTSIZE-CAPACITY 0)
(CRITICAL-CAPACITY 0) (LOWER-LOBE CAPACITY 0)
(PAX-CAPACITY 0) (ACCOMPANYING-CAPACITY 15)
(CONSTRAINT-1 10) (CONSTRAINT-2 30) (CONSTRAINT-3 50)
(CATEGORY-CODE NIL) (DESIGNATION MILITARY))
```

Figure 2.1. Aircraft Object Representation

aircraft capacities for different types of cargo, true air speed, range, and onload and offload times. MACPLAN maintains aircraft objects for all types of aircraft available to the planner. When selected by the planner, the aircraft objects are placed into a force package and saved in a planset along with the requirements. A representation of a sample aircraft object is shown in Figure 2.1.

The requirements are lists containing amounts of cargo which must be moved from one station to another within specified periods of time. These requirements include data such as load designator, onload station, offload station, earliest available date, earliest arrival time, latest arrival time, and cargo-listings. A representation of an example requirement is shown in Figure 2.2.

The load designator is simply a number, such as R15, used to uniquely identify each requirement. The onload-station and offload-station fields are actual station objects in MACPLAN. The onload-station specifies where the cargo listed in the requirement is to be loaded onto an aircraft, or the source, and the offload-station

```
($F (LOAD-DESIGNATOR R24) (ONLOAD-STATION KSLC)
(OFFLOAD-STATION EGUL) (AVAILABLE-TIME C000)
(EARLIEST-ARRIVAL-TIME CC005) (LATEST-ARRIVAL-TIME C007)
(PRIORITY 001) (BULK-CARGO 0) (OVERSIZE-CARGO 0)
(OUTSIZE-CARGO 15) (PAX 0) (MIN-LAUNCH-INTERVAL "0030")
(MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD ENGINE-RUNNING)
(ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE
NONE) (ACFT-CATEGORY-CODES NIL))
```

Figure 2.2. Requirement Object Representation

```
($F (ICAO KFFO) (NAME "WRIGHT-PATTERSON AFB") (COUNTRY-CODE
39) (GEO-CODE ZHTV) (LATITUDE "39 49N") (LONGITUDE "84 02W")
(STATION-LOGISTIC-PERMISSION NIL) (CONSTRAINT-TYPE 1)
(CONSTRAINT-FACTOR 100) (STERILE-START "0000") (STERILE-
STOP "0000") (INTERVAL-TYPE ARRIVAL) (MAX-GROUND-TIME "0830)
(ARRIVAL-DEPARTURE-INTERVAL "0030") (NUMBER-ACFT-PERMITTED-
PER-INTERVAL 1) (AERIAL-PORT-DESIGNATOR P)
(SPECIAL-DESIGNATOR ENGINE-RUNNING) (LOGISTIC-CODE NIL)
(CONSTRAINT-CARGO 999999) (CONSTRAINT-PAX 999999) (TYPE AFB))
```

Figure 2.3. Station Object Representation

specifies where the cargo needs to be transported to, or the destination. Each station object contains all necessary information about the particular ground station specified. This information includes the ICAO code (four-letter station designator), name of the airport, country, geographic code, latitude, and longitude. A station object representation is shown in Figure 2.3.

Other objects in MACPLAN include groups, paths, links, permissions, and UTE&JSCP tables (utilization rate for the aircraft fleet). Groups are lists of stations that are close together in distance. For example, all stations in the eastern United

States might be contained in a group called Eastern-US. Paths are lists of groups specifying the path followed by a requirement from its source to its destination. They can also include enroute groups if the distance from the source station to the destination station is too great to make in one flight. Links are segments of a path linking one station or group to another station or group. Paths among different groups may share a common link, but each path is between only two groups. Permissions contain the characteristics of a given station such as maximum size of aircraft that can be handled on the runway or the maximum number of aircraft allowed on the ground simultaneously. The UTE&JSCP objects specify the UTE, or utilization rate of the aircraft. If a type of aircraft can only be flown ten hours a day on average, the UTE rate for that aircraft is listed as ten. This forces MACPLAN to allot a realistic number of ton-miles per day to each aircraft when computing airlift capability.

The Planning Process with MACPLAN

A planner begins the planning operation with a set of requirements. These requirements may be troops, jeeps, tanks, or any other type of cargo which must be moved from one location to another. Most plans include large amounts of cargo from many different locations and may span several months. These requirements are loaded into MACPLAN, which takes each requirement and adds it to a list of cargo arriving and leaving each station. The cargo originating at that station is kept in a delivery-list and the arriving cargo is stored in an arrival list. The delivery list is broken down into lists of cargo going to different stations.

The requirements are also broken down into a normalized list, called norm-delivery-list and a list of tons, called delivery-list. The norm-delivery-list is a list of the 1000-ton-miles per day and the delivery-list is simply a list of the tons of cargo which need to be moved. All cargo is listed in tons except for passengers which is listed in the number of passengers. The normalized passenger requirement is in

1000-troop-miles per day.

After loading the requirements, the planner forms groups from the stations listed in the requirements. This can be done manually by the planner or automatically by MACPLAN, which forms the groups by country. All stations in England will be in their own group, but for large countries like the United States the groups are divided by regions such as southwest or northeast. These groups are designated as to-from groups because the requirements are either going to or from one of these groups to another. MACPLAN considers all stations in a group as one when considering paths or distances from one group or station to another. The creation of groups allows MACPLAN to consider fewer objects when creating paths between stations. The paths are from group to group instead of from station to station.

After forming to-from groups, the planner selects aircraft to move the requirements. MACPLAN does not offer any help in selecting the aircraft or the operating units used to move certain cargo. It allows the planner to create a force package without any C-5 aircraft even if the requirements include outsize cargo and a C-5 is the only aircraft capable of transporting outsize cargo. After selecting a type of aircraft, such as C141-B, the planner must select an operating unit for the aircraft. MACPLAN provides a list of all operating units for each specific type of aircraft for the planner to choose from.

The next step in assembling a planset is creating paths to be flown by the aircraft selected in the previous step. Paths can be automatically generated by MACPLAN between all groups in the data base. Once the paths are generated, range checks must be performed to insure that the aircraft selected are able to fly the distances between groups. If an aircraft selected is not capable of flying the range necessary, enroute groups or stations must be added to the path to break the flight into shorter lengths. MACPLAN does not create separate paths for aircraft with longer ranges. All aircraft selected must be capable of flying all paths created or a range violation is generated.

Once the range checks are completed and all paths are of a suitable distance, the number of aircraft needed each day must be chosen. The planner selects the number of aircraft to be provided by each unit. If there are two different operators of C-5s, the planner has to decide how many planes each unit will provide.

After selecting the number of aircraft for each day the planner performs a capacity check against the requirements. This creates a graph comparing airlift capacity with requirements showing any backlog of unmoved cargo. This backlog is computed by comparing the normalized requirements, which are in units of 1000-ton-miles per day, with the normalized airlift capability. If there is an unacceptable backlog or surplus of aircraft, the planner may change the staging of aircraft to eliminate the problem.

When the planner is satisfied with the force package selected to satisfy the requirements, it is saved into a planset. The planset includes the force package and operating units as well as the paths to be flown. This planset is sent to FLOGEN to generate a schedule from the supplied information to determine how well the requirements can be moved with the specified resources. FLOGEN takes a planset from MACPLAN and generates a day to day schedule with the planes and cargo involved by forward discrete-event simulation. This schedule takes into account all of the constraints for each plane and station. The resulting schedule is similar to an airline schedule with exact takeoff and landing times for each aircraft in the plan. These schedules are manually analyzed for inefficiencies or bottlenecks and the plan is modified accordingly. This cycle is repeated until a satisfactory plan is created.

Current Analysis Capabilities

The analysis capabilities of MACPLAN are limited to range checking, capacity checking (also called workload estimation), and a preflow analyzer. The range checker simply checks the routing network for any paths too long for an aircraft to fly. The capacity checker compares the normalized requirements against the normal-

ized airlift capability to provide a backlog graph. The preflow analyzer is designed to be a FLOGEN simulator, but did not work on the release of MACPLAN used.

The capacity checker in MACPLAN compares the normalized requirements against the normalized airlift capability. When computing the normalized requirements, MACPLAN takes each type of cargo in each requirement and divides this by the number of days in the delivery window for the requirement. The delivery window is computed by subtracting the available date from the latest arrival date and adding one. This gives the number of days which are available for transporting this requirement. For example, if a requirement is available on day 5 of the plan and must be delivered by day 7, the delivery window is three days. The number found by dividing the tons of cargo by the delivery window is the number of tons (or passengers) which must be moved on the average for each day the requirement can be moved. This number is multiplied by the distance between the onload and offload stations of the requirement. This gives the number of ton-miles per day which must be moved. This number is divided by 1000 to produce a normalized figure of 1000-ton-miles per day. The normalized requirements are then added together for all of the requirements which are on the same days to produce a normalized requirements list for each day of the plan.

The airlift capability is produced by adding together the capacities of all aircraft which are selected for each day. The totals are computed for each type of aircraft and then added together for each type of cargo to produce an airlift capability for each day. The cargo capacity of each type of aircraft is multiplied by the number of aircraft sourced for that day, the UTE rate for the aircraft (in hours), the true air speed of the aircraft (in miles/hour), and a round trip factor and then divided by 1000 miles. The factor is 0.47, which accounts for each aircraft having to make at least half of each round trip empty.

After computing the normalized requirements and airlift capability, the backlog for each day is found by subtracting the requirements from the airlift capability. If

there is a surplus of airlift on any day and a backlog of requirements, the surplus is used to deplete the backlog. This produces an estimate of how many requirements can be moved on each day by the force package selected and an estimate of when each requirement can be moved to its destination. This estimate is optimistic because it assumes each aircraft operates at full capacity for each flight.

III. TEMPORAL CONSTRAINT SYSTEMS AND CONSTRAINT PROPAGATION

Introduction

Deciding when to perform certain actions to make the most efficient use of available time has always been a problem. Using computers to reason about time can significantly increase the number of choices that can be analyzed. Several methods of reasoning about time are suitable for application to airlift planning analysis. These methods are temporal constraint propagation and constraint processing clustering schemes. Temporal constraint propagation allows constraints between events to propagate through a network and to determine relationships among related events in time. Using a temporal reasoning system to constrain these events allows multiple sequences of events to be searched to find the most efficient sequence. Clustering schemes simplify the constraint propagation in large networks by limiting propagation to occur only between specified events. The time required to evaluate airlift schedules can be dramatically reduced using these techniques.

Temporal Constraint Networks

Introduction. Temporal constraints represent bounds on the time that passes between two events. A network of temporal constraints expresses the possible times that events may occur relative to other events in the network. For example, suppose it takes between 10 and 20 minutes to take a shower, between 15 and 20 minutes to eat breakfast and between 25 and 45 minutes to get to work. If you get out of bed at 6 A.M., you will arrive at work sometime between 6:50 and 7:25 A.M. Any known information constrains the possible event times even further. If your shower takes only 10 minutes, your arrival time is now limited to between 6:50 and 7:15. Dechter et al. (2) describe a formal method to represent events and build a temporal reasoning system to solve such networks in polynomial time.

The temporal reasoning system described consists of a temporal knowledge base, a routine to check its consistency, a query answering mechanism, and an inference mechanism capable of discovering new information. The knowledge base contains propositions to which temporal intervals are assigned. A proposition may be "I was driving the car" or "the book was on the table" with each interval representing the time period during which the corresponding proposition is true. The temporal information may be relative (I had breakfast before I took a shower) or metric in nature (I slept for exactly 8 hours). Placing constraints on the beginning and ending time-points defining an interval during which a proposition is true provides a means of expressing temporal information about the proposition.

If X1 and X2 are two time-points, the temporal distance between them may be expressed as $X1 - X2 \leq c$, where c is the maximum time that can elapse between the two events. Expressing the distance between multiple time-points gives us a set of linear inequalities on the time-points under consideration. If the time-points X1 and X2 represent the interval corresponding to the proposition "John was going to work" and we know John rides the bus which takes between 30 and 40 minutes to get John to work, the inequality

$$30 \leq X2 - X1 \leq 40$$

represents this interval. Disjunctions must also be represented in the system. Suppose John could also carpool to work which takes between 45 and 50 minutes. This interval would then be expressed as the set of inequalities

$$30 \leq X2 - X1 \leq 40 \text{ or}$$

$$45 \leq X2 - X1 \leq 50.$$

The temporal reasoning system must be able to answer such questions as "Is it possible that John left at 7:00 and arrived at work at 7:45?" or "If John arrived

at work at 8:00, when could he have left his house?" A formal representation of this problem based on temporal constraint satisfaction is introduced in the paper with two methods of arriving at a solution. The first method is decomposition of the temporal network and the second is using a relaxation algorithm to solve the constraint network.

A Temporal Constraint Satisfaction Problem Model. A formal representation for a temporal constraint satisfaction problem (TCSP) is described below. A temporal constraint satisfaction problem (TCSP) involves a set of temporal variables representing time-points and unary and binary constraints on these variables expressed in terms of temporal distance. A binary temporal constraint, $T_{12} = (a, b)$, between the variables X1 and X2 indicates the permissible values for the temporal distance X2 - X1, expressed by the inequality

$$a \leq X2 - X1 \leq b.$$

A unary temporal constraint, $T_1 = (a, b)$, on X1 indicates permissible values for the occurrence of X1, expressed as the inequality

$$a \leq X1 \leq b.$$

A network of temporal constraints can be represented by a directed temporal constraint graph, whose nodes represent temporal variables and whose edges represent constraints on the temporal distance between the variables. Figure 3.1 shows a sample temporal constraint network expressing the example described earlier about John going to work. Node 1 represents John getting up, node 2 represents finishing the shower, node 3 represents finishing breakfast, and node 4 represents arriving at work. A solution to this network is a set of values which can be assigned to the edges between the nodes and which satisfies all constraints. Assigning the values 15,

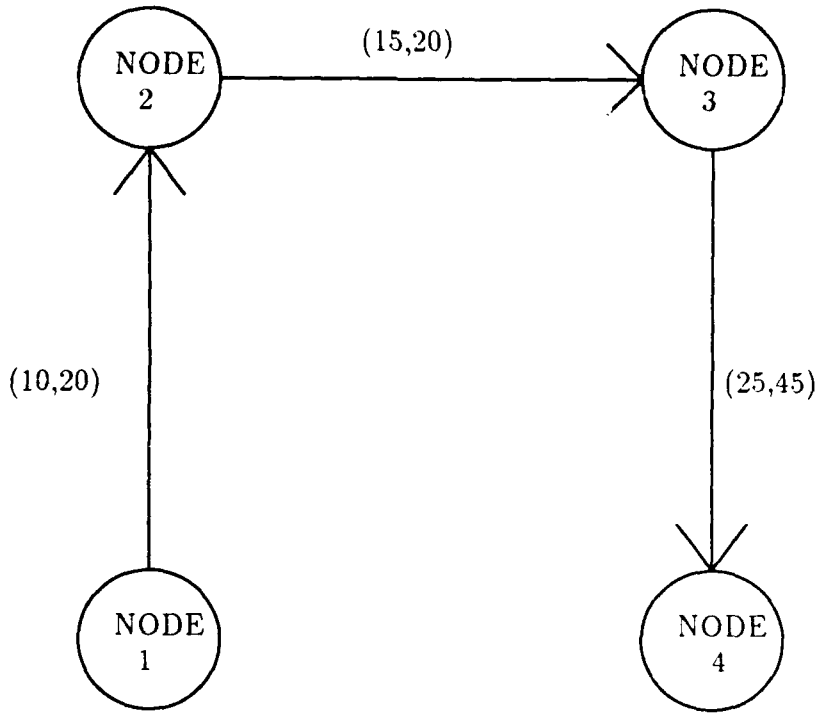

Figure 3.1. Sample Temporal Constraint Network

18, and 30 to the edges between nodes 1–2, 2–3, and 3–4 respectively is one possible solution to this network.

A feasible value is defined as any valid value for an edge in a solution set and the set of all feasible values for a given variable is its minimal domain. A network is consistent if at least one solution exists. The binary operations union, intersection, and composition are defined for temporal constraints.

The union of two constraints, represented by $T \cup S$, consists of any values allowed by either T or S. Intersection of two constraints, $T \cap S$, allows only the values which are allowed by both T and S. Composition of two constraints, $T \otimes S$, allows only the values (a,b) for which there is at least one value c such that (a,c) is in T and (c,b) is in S. Constraint T is said to be tighter than constraint S if every pair of values allowed by T is allowed by S. Two constraints are equivalent if they

represent the same set of solutions.

Another important property of constraint networks is decomposability (2). A network is considered decomposable if and only if every variable can be assigned any value allowable by the constraints on that variable and permit a solution to the network. This property allows backtrack-free search in finding a solution. In other words, once a variable is assigned a value, a solution can be found without changing the value.

Given a network of binary constraints, the first problem is to determine if the network is consistent, i.e., if a solution exists. If a solution exists, the minimal domain of each variable should be found (find all possible values for each variable). Another problem may be to find the relationship between two variables. Two approaches to solving these problems are discussed in the next sections.

The Simple TCSP. A simple temporal constraint satisfaction problem (STCSP) is one in which all constraints have a single disjunct, i.e., each bound has only one possible interval. Since all constraints are a single interval, these problems can be solved in polynomial time by solving the set of inequalities associated with the network. However, a simpler graph-based algorithm can be used to solve this class of problems. The graph representation discussed is based on a distance graph. Each node represents a temporal variable and the edges connecting the nodes represent the maximum value of the time bound between the two nodes. Each edge in the graph indicates a bound on the interval between the times of the original and ending nodes. A negative bound implies that the originating node occurs after the ending node. A distance graph showing John's morning schedule is shown in Figure 3.2. The nodes represent the same events as in the earlier example. The distance graph indicates that John's shower is over between 10 and 15 minutes after his getting out of bed. Summing up the edges from node 0 to node 4, we can determine the maximum time after getting out of bed that John arrives at work. Summing

up the edges from node 4 to node 1 determines the maximum time after arriving at work that John gets out of bed. This number is negative, which means John gets out of bed at least this amount of time before arriving at work. These two sums provide an interval in which John arrives at work after getting out of bed.

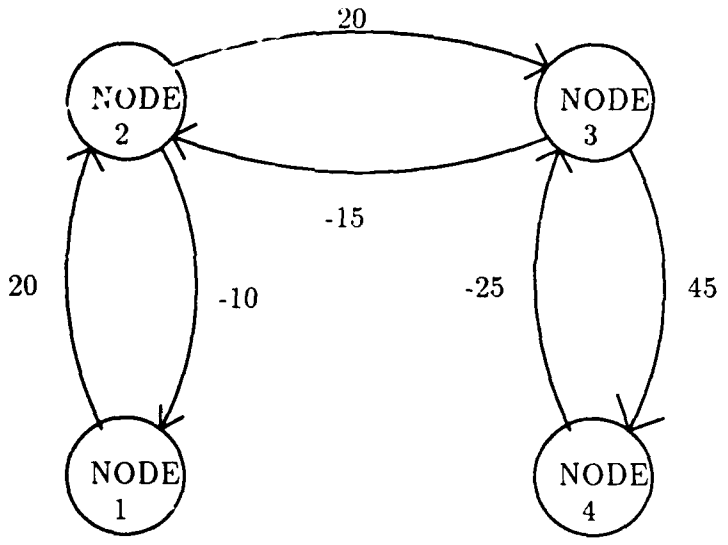

Figure 3.2. Time Distance Graph

Dechter et al. (2) present and prove several theorems. The first theorem states that a distance graph is consistent if and only if it contains no negative cycles. The second is that any consistent simple TCSP is decomposable relative to constraints specified by its distance-graph representation. Theorem 2 provides an efficient algorithm for finding a solution to a simple TCSP. Since the TCSP is decomposable, we can assign any value satisfying the distance graph constraints to each variable. The domains characterized by the distance graph are also minimal for the TCSP. Proof

of these theorems is discussed in the referenced paper. This problem can now be solved by applying Floyd-Warshall's all-pairs-shortest-paths algorithm (2). This algorithm runs to completion in time $O(n^3)$ and negative cycles are easily found. This is a polynomial time algorithm for determining the consistency of a simple TCSP, finding a solution, and determining the minimal domains and minimal network.

A Decomposition Method for Solving the TCSP. Dechter et al. put forth a method to solve a general TCSP by decomposing it into several simple TCSPs, solving each one, and combining the results (2). Given a network of binary temporal constraints, T, a labeling of T is defined as a selection of one interval from each constraint. This method allows for disjunctive constraints with different labellings. Each possible labeling represents a simple TCSP which can be solved by the method described in the previous section. The TCSP is consistent if at least one of the labelings of the network is consistent. The minimal network of T can be determined by finding the union of all simple TCSPs which are consistent. The complexity of solving a TCSP in this manner is $O(n^3k^e)$, where k is the maximum number of disjunct intervals of an edge and e is the number of edges in the constraint graph. Although this is worst-case complexity, several techniques can be employed to reduce the required computation in many cases.

A Relaxation Method for Solving the TCSP. The decomposition method suffers from two drawbacks. First, the techniques used to solve the network do not exploit the fact that each labeling differs from other labelings by only a small number of constraints. Each labeling is solved from the beginning with no computational savings even if it is almost identical to the previous labeling. Second, the process of translating each labeling into a distance graph may be cumbersome in practice. An alternative method for solving the TCSP, applicable directly to the original constraint graph, is discussed in the following sections.

The all-pairs-shortest-paths algorithm discussed in the previous section can be

considered a relaxation algorithm: at every step the value of an edge is updated by an amount depending only on the current values of adjacent edges. A relaxation algorithm that enforces path consistency on TCSPs is described in the paper. A path consistency algorithm was described by Dechter et al. (2) and shown to be identical to applying Floyd-Warshall's all-pairs-shortest-path algorithm to the distance graph of a TCSP.

The algorithm discussed was put forth as an attempt to simplify the computation time to solve a TCSP. There are several questions still unanswered about the algorithm. Although not proven by Dechter et al. they believe the algorithm converges to a solution in an efficient amount of time.

Constraint Propagation

Planning and scheduling problems typically have numerous constraints from a variety of sources. Constraints can come from the availability of resources or the limitations of the resources used. One method used to solve such problems is constraint propagation, sometimes called relaxation (7). Constraint Propagation systematically eliminates values which are not possible based on multiple constraints until a solution is found. A constraint propagation problem is specified by a set of variables and a set of constraints limiting the values the variables can take on (5). Specifying a value for one variable may limit the possible values for other variables. A solution to a constraint propagation problem is a set of values which does not violate any constraints.

Constraint propagation arrives at a solution by choosing a set of values for one variable and propagating those values to all other constraints involving that variable, eliminating any values which do not satisfy all variables simultaneously. Repeated applications of this method to all variables either arrives at a possible solution or eliminates all choices for a variable. If all possible choices for a variable are eliminated, there is no solution to the problem.

An example constraint propagation problem is the cryptarithmetic problem where letters are used to represent numeric digits in a mathematical equation. One such problem is listed below:

```
  S E N D
+ M O R E
---------
M O N E Y
```

A solution to the problem is found when a digit is substituted for each occurrence of a letter and the resulting equation is mathematically correct. To solve this problem with constraint propagation, we can let each letter have a set of possible digits which they could represent. We would then represent each part of the problem as a constraint. One constraint would be that **S** + **M** + 0 or 1 = **MO**. The 0 or 1 represents the carry from the column preceding **S** and **M**. By using all constraints, values for each letter can be eliminated until all values left satisfy the constraints. All letters can be any digit from 0 to 9 except the letters **M** and **S**. These letters cannot be 0 because they are in the beginning of the numbers and numbers don't start with 0. We must also restrict the letter **E** to be a 5 if we want a single solution to the problem. If we let C1 represent the carry from the ones column, C10 represent the carry from the tens column and C100 represent the carry from the hundreds column, the possibilities for each variable are shown below.

```
M: [1,2,3,4,5,6,7,8,9]
S: [1,2,3,4,5,6,7,8,9]
O: [0,1,2,3,4,5,6,7,8,9]
E: [5]
N: [0,1,2,3,4,5,6,7,8,9]
R: [0,1,2,3,4,5,6,7,8,9]
D: [0,1,2,3,4,5,6,7,8,9]
Y: [0,1,2,3,4,5,6,7,8,9]
C1: [0,1]
```

```
C10: [0,1]
C100: [0,1]
```

The constraints effected by the equation are as follows:

```
1:  D + E = Y + (10 * C1)
2:  N + R + C1 = E + (10 * C10)
3:  E + O + C10 = N + (10 * C100)
4:  S + M + C100 = O + (10 * M)
5:  Each letter is a unique digit
```

By enforcing multiple constraints involving the same variable and eliminating any possible values which do not satisfy all constraints, we can arrive at a solution. For example, if we examine variable **M** first, constraints 4 and 5 mention **M**. The only value of **M** satisfying constraint 4 is 1, since 0 is not in **M**'s original possibility list, and any value of **M** greater than 1 would make the right side of constraint 4 at least 20, and there is no combination of **S** and **C100** drawn from their possibility lists whose sum plus 2 could be 20 or greater. We then select another variable and eliminate values for it based on the possibilities of **M** already made. This process is repeated for any variable with more than one possibility until a solution is reached. The solution is shown below.

```
  9 5 6 7
+ 1 0 8 5
---------
1 0 6 5 2
```

Constraint propagation can be applied to scheduling problems also. For example, we have constraints on our daily schedule limiting the times when certain events may occur. These constraints are imposed on us by the physical world around us as well as by other people's actions. One such constraint is the time required to

travel from home to work. By examining all of these constraints, we can arrive at a workable schedule to accomplish our goals for each day. Although we don't normally think of our daily schedule in such terms, these constraints must be considered if we want a computer to generate a schedule for us.

Clustering Schemes

Constraint propagation reaches all time-points in a reference set. Deciding how to cluster the time-points into reference sets has an impact on the time required to propagate new constraints. If too many events are in each reference set, the computation time increases to a point where performance is not satisfactory. With too few events in each reference set, the system maintains fewer durations between time-points and must rely on slower or less precise algorithms to calculate the constraints on points in different reference sets.

There are several methods available for determining the clustering of events into reference sets. Clustering based on the temporal relationships between events is the simplest, but may not provide optimal performance. Dechter and Pearl describe a tree-clustering scheme based on transforming a constraint graph into a tree structure (3). Kohane describes an automatic method for clustering using heuristics to actively change the clustering based on performance and the frequency of past queries (6).

Clustering based on relationships may be as simple as placing all events relating to a certain object in the same reference set. This method is certainly simple to implement, but the performance is related directly to the number of events related to each object. The efficiency of a system clustered with this algorithm would most likely be less than optimal.

The clustering scheme described by Dechter and Pearl transforms any constraint graph into a tree structure by removing redundant paths through the graph. Each branch of the tree can then be clustered together for constraint propagation within that cluster. This method is more complex than the previous one and again,

the performance is related to the resulting tree structure. While providing improved performance over no clustering, there is no guarantee that the optimal performance is obtained.

Kohane's automatic clustering method monitors the performance of the system, and when it degrades below a certain level, the system activates some heuristics to cluster the points into reference sets to improve the system's efficiency. These heuristics are based on the frequency of queries between points. The points which have the most queries between them are placed into reference sets together. This method, called "performance driven clustering," always provides a specified level of performance (if possible).

IV. Airlift Planning Analysis System

Introduction

The airlift planning analysis system designed in this thesis consists of two parts: a temporal reasoning system and an airlift capacity analyzer. The temporal reasoning system maintains information about times required to perform specific actions in the plan and the capacity analyzer provides information constraining the times between events. Together these two systems provide a method to analyze preliminary airlift plans.

General Approach

The temporal reasoning system maintains information about time relationships between events, or time-points. When any information constraining the time relationship, or duration, between events is discovered, the system enforces this constraint on the events. If this information also constrains relationships with other events, it is propagated to all affected events. For example, if we know it takes five hours for a plane to fly from Los Angeles to New York and the plane departs Los Angeles at 10:00 A.M. PST, it is not possible for the plane to arrive in New York until at least 3:00 P.M. PST. The departure time plus the flight time place a constraint on the the arrival time in New York. If the plane were continuing on to London, we could also place a constraint on the earliest time for a London arrival.

Analyzing a set of airlift requirements provides a set of constraints on specific events in the plan. Further constraints arise from examining the aircraft and operating units involved. The number and capacity of the planes involved limit how much cargo can be transported on a given day.

Placing constraints gained from analyzing requirements and resources into the temporal reasoning system provides information about when the plan may be com-

pleted. This chapter describes the temporal reasoning system and the capacity analyzer and how they work together to provide information about a suggested plan.

Temporal Reasoning System

An integral part of this project is the temporal reasoning system. This system is capable of maintaining information about relationships among events or points in time. After entering known constraints about the relationship between events, the system provides a possible interval in which one event may occur with respect to the other. Two time-points must be connected within the network to obtain a possible relationship between them. The temporal reasoning system developed for this project is based on the formal representation discussed in Chapter 3. There are two types of objects in the system: time-points and durations. The time-points represent events and the durations represent the time between events.

Design

The first object in the system, the time-point, is made up of a name, a list of durations containing this time-point, a list of in-durations containing this time-point, and a reference set. A sample time-point is shown in Figure 4.1. The name of the time-point is used for debugging purposes and to query the system about certain time-points. The list of durations contains the duration objects (described in the next paragraph) that have this time-point as their beginning. The in-durations list contains duration objects that have this time-point as their end. The reference set is an identifier clustering a set of time-points that are related in some way. All time-points in a reference set contain durations to all other time-points in that reference set.

The second object in the system, the duration, contains a beginning time-point, an ending time-point, and a list of bounds. A sample duration is shown in Figure 4.2. The list of bounds may contain only one bound (representing a single

```
NAME:            SAMPLE-TIME-POINT
DURATIONS:       (LIST OF DURATION OBJECTS)
IN-DURATIONS:    (LIST OF DURATION OBJECTS)
REFERENCE-SET:   (LIST OF REFERENCE SETS)
```

Figure 4.1. Sample Time-Point Object

```
TIME-POINT-1:    BEGINNING TIME-POINT OBJECT
TIME-POINT-2:    ENDING TIME-POINT OBJECT
BOUND:           (LIST OF BOUNDS)
```

Figure 4.2. Sample Duration Object

interval) or multiple bounds (representing multiple intervals) limiting the time that can elapse between two time-points. If more than one bound is present, the opposite duration (from the ending time-point to the beginning time-point) must contain the same number of bounds. These corresponding bounds define the possible time intervals between the time-points and must not overlap. Time-points are connected by durations which maintain the upper bound of the time that the ending time-point is allowed to occur after the beginning time-point. For example if a duration existed between two time-points, T1 and T2, with an upper bound of 30 minutes, we would say T2 will occur no more than 30 minutes after T1. An interval can be established in which T2 must occur after T1 by asserting a duration from T2 to T1 of -10 minutes. These two durations would limit T2 to an interval of between 10 and 30 minutes after T1. It is possible to constrain the time between events to disjoint intervals. For example, T2 could be constrained to occur either between 10 and 30 minutes after T1 or between 45 and 60 minutes after T1. If these constraints were imposed on the system, the query ***(Interval-constraint T1 T2)*** *would return the* list **((10 30) (45 60))** as the possible intervals.

The relationships among all time-points within a reference set are always maintained by the temporal reasoning system. When a new bound is asserted on two time-points in a reference set, it is propagated between the two original time-points and all other time-points in the reference set by the functions **Propagate-Forward** and **Propagate-Backward**. The reference set limits the propagation to time-points within a single reference set. When the propagation routine encounters a time-point outside the reference set of the original time-point, the propagation halts. Some time-points are in more than one reference set to insure that the durations are maintained for the boundary time-points between reference sets. This allows intervals to be found between time-points in different reference sets.

Using reference sets reduces the computational complexity of propagating bounds through the network. Dividing the time-points into reference sets provides less information stored in the data base, but decreases the time required to assert information into it. The time savings are much greater than the bounds lost due to using reference sets (6). Without reference sets, the number of bounds maintained by the system grows with the square of the number of time-points. Using reference sets decreases this growth to a linear relationship.

Figure 4.3 shows a sample network involving a flight from Los Angeles (LA) to New York (NY) and continuing on to London (LON). The nodes represent takeoff and landing at each city and are in the same reference set. The time spent on the ground is ignored to simplify this example and takeoff is immediately following landing in NY. The current constraints are 5 hours from LA to NY and 9 hours from NY to LON. The current constraint on the time from LA to LON is the sum of the constraints in between, 14 hours. If the winds between LA and NY are favorable and cut the flight time to 4 hours, the new constraint would then propagate through to the LA to LON constraint and decrease the total to 13 hours. The system asserts and maintains the duration from LA to LON because they are in the same reference set even though a direct assertion was not made between these events.

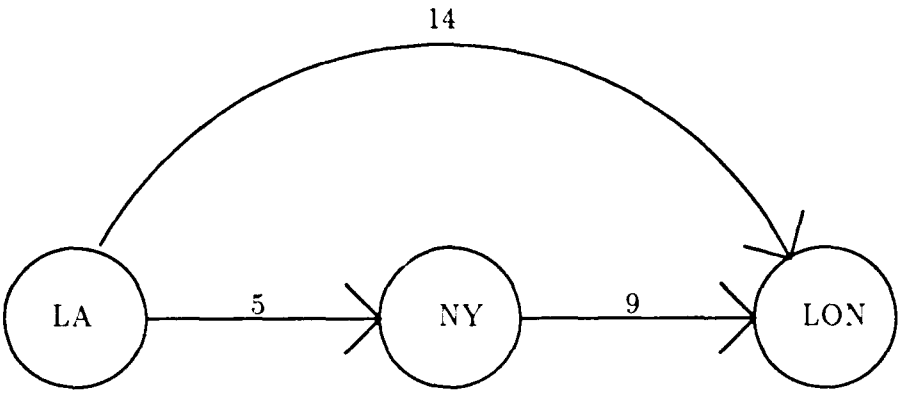

Figure 4.3. Propagation Example

Propagate-Forward uses the Duration-List of the time-points it encounters to propagate the new bound through the entire reference set. The new bound is added to the next bound and if the resulting bound is more restrictive than the previous one; it is changed and the propagation continues. In the above example, the new bound from LA to NY (4 hours) is added to the existing bound from NY to LON (9 hours) and the result (13 hours) is less than the current bound from LA to LON. Therefore, the existing bound is changed to the new bound (13 hours). If another city was present after LON, the new bound (13 hours) would be added to the next existing bound to determine if it restricted the bound from LA to the next city. The system calculates and maintains the bounds between all time-points in a reference set even though these bounds may not be explicitly asserted. This process continues until the resulting bounds no longer restrict the current bounds. Propagate-Backward operates in the same manner, but uses the In-Duration-List of the time-points.

Operation

A temporal network can be built using time-points and durations. By using the function Create-Time-point to create all desired time-points in a network and then using the functions Assert-interval, Assert-Not-Interval, and Assert-Duration to constrain the time-points, we can build a temporal network. The relationships

between all time-points in a reference set are maintained by propagating the bound through the entire reference set when a new bound is asserted between two time-points in the reference set.

The function Create-Time-point creates a new time-point. The format for this function is:

(create-time-point name reference-set)

The parameters required by Create-Time-point are a name and a reference set for the time-point. A new time-point is created with the name and reference set given as parameters and with a duration list and in-duration list of NIL.

Assert-Duration asserts a bound on only one of the durations between two time-points. The format for this function is:

(assert-duration T1 T2 bound)

This assertion restricts the time between T1 and T2 no more than the value of bound. After asserting the new bound, it is propagated through the system to update all bounds which may be affected. If this bound cuts off an entire interval from the current bounds, the reverse duration is changed accordingly. For example, if the current possible intervals were **((10 30) (45 60))**, and we assert a new upper bound of 35, the resultant possible interval would then be **((10 30))**.

The function Assert-Interval asserts two durations which constrain the second time-point to occur within the given interval after the first time-point. The format for this function is:

(assert-interval T1 T2 B1 B2)

B1 is the lower bound on the interval and **B2** is the upper bound. The function assert-interval actually asserts two bounds between the time-points using the Assert-Duration function:

(assert-duration T1 T2 B2)

(assert-duration T2 T1 -B1)

The first bound is a lower bound which gives the earliest time after the first time-point at which the second time-point can occur. A negative bound allows the second time-point to occur before the first time-point. Tl second bound is the largest amount of time that can lapse between the first time point and the second time-point. If both bounds are negative, the second time-point must occur before the first time-point. If the first bound is negative and the second is positive, the second time-point may occur either before or after the first time-point; if both bounds are positive, the second time-point is forced to occur after the first one.

Assert-Not-Interval asserts the proper bounds to insure that the second time-point does not occur within the two given bounds after the first time-point. The format for this function is:

(assert-not-interval T1 T2 B1 B2)

This function asserts two bounds on the durations between the two time-points. The first bound asserted is on the duration from T1 to T2 and is equal to bound 1. If bound 1 is less than the current bound on this duration and bound 2 is less than the current bound from T2 to T1, then bound 1 is added to the duration from T1 to T2. The new bound in this duration is a list containing two bounds and the reverse duration must also contain two bounds on the interval. For example if the current bounds limited the interval between T1 and T2 to between 10 and 30 minutes, the duration from T1 to T2 would have a bound of (30) and the reverse duration would have a bound of (-10). If we then used assert-not-interval with bounds of 15 and 20, the new interval between the two time-points would be ((10 15) (20 30)). This means T2 will occur either between 10 and 15 minutes after T1 or between 20 and 30 minutes after T1. If one of the new bounds is outside the current interval, then that duration would not change and the other duration would change to the new bound.

The system may be queried as to the interval between any two time-points by using the function interval-constraint. The format for this function is:

(interval-constraint T1 T2)

This returns a list containing the bounds of the possible intervals within which T2 may happen with respect to T1.

Temporal Network Operation

For each requirement, time-points are created for each important event occurring in the transportation of each type of cargo in the requirement. A time-point is also created for the beginning of the plan and the end of the plan. The time-points created are called BEGIN-PLAN, AVAILABLE-CARGO-RX, ONLOAD-CARGO-RX, LAUNCH-CARGO-RX, LAND-CARGO-RX, OFFLOAD-CARGO-RX, and END-PLAN. There are events for each possible type of cargo: bulk, oversize, outsize, and pax (for passenger). The X in the RX stands for the load-designator of each requirement such as R1 for the first requirement. The resulting time-points for the bulk cargo in requirement R1 would be AVAILABLE-BULK-R1 through OFFLOAD-BULK-R1. For each requirement, durations are asserted from BEGIN-PLAN to the AVAILABLE-CARGO-RX time-point and so on to the END-PLAN time-point. An upper bound of infinity is used and is asserted as a bound of nil, which represents no information on the bound.

A temporal network is built by using the function **Analyze-plan begin end** where begin and end are calendar days specifying the interval to analyze the plan. They must be in the format **'CXXX** where **XXX** specifies the day relative to the beginning of the plan. After asserting the network, the MACPLAN functions for displaying the backlog graph are called with the information gathered by the capacity analysis system. When the system starts, a duration is asserted from **Begin-Plan** to each of the **Available-Cargo-RX** time-points with a bound computed from the rules below. The capacity analysis system (described in the next section) is then

called with the requirements for the first day of the plan. When a requirement is completely moved as determined by the aircraft available, a duration is asserted from the **Available-cargo-RX** time-point to the **Onload-Cargo-RX** time-point based on the aircraft selected by the capacity analysis system. Part of a requirement may be moved on one day with the rest of it remaining until several days later. These cases are not asserted as moved until the entire cargo has been assigned to an aircraft and classified as moved by the system. The next event in transporting the requirement is the takeoff, which is calculated by the rules below. A flight time is then calculated for the requirement based on the origin and destination stations of the requirement and the selected aircraft. This time is asserted as the duration from **Takeoff-Cargo-RX** and **Landing-Cargo-RX**. The type of aircraft selected is then used to determine the offload time required and is asserted from the **Landing-Cargo-RX** time-point to the **Offload-Cargo-RX** time-point. A final assertion is made from the **Offload-Cargo-RX** time-point to the **End-Plan** time-point. The beginning and ending time-points of the temporal network are now constrained by the times required to move each of the requirements. The shortest possible time in which the entire plan can be finished is found by the command **(Interval-Constraint Begin-Plan End-Plan)**. The earliest time that any one requirement is moved can be found by finding the longest time of moving each of the four components of the requirement. If a requirement does not have any bulk cargo, there are no restriction on those time-points.

The following constraints are used to post the durations between the time-points created for each requirement in the database.

Begin-Plan to Available - The duration from begin-plan to available for each requirement is bound by the time listed in the requirement as the available time and infinity. The time in the requirement is listed in relative calendar time, such as C004, denoting day 4 of the plan, and is converted to minutes to the earliest time of the day listed. The "days-to-minutes-earliest" function is used to convert

this time. The upper bound of infinity denotes no information on the longest time required to perform this action.

Available to Onload - The available to onload duration is constrained by the onload time for the aircraft used to carry the cargo listed in the requirement. The shortest onload time is found by taking the shortest onload times for all planes compatible with each different type of cargo and then taking the longest of these onload times. This represents the shortest amount of time in which the cargo in this requirement can be onloaded. An upper bound of infinity is used here also.

Onload to Launch - The onload to launch duration is constrained to be between 0 and infinity. In other words, launch may be any time after onload. This duration could be limited if more information about the time required to gain takeoff clearance and perform all pre-takeoff functions were known.

Launch to Land - the duration from launch to land is constrained by the time required for the selected aircraft to fly from the onload station to the offload station using a path selected from the planset. The ground time at enroute stations and prevailing winds are not accounted for in the present system. An upper bound of infinity is used.

Land to Offload - The duration from land to offload is constrained to be between the offload time for the selected aircraft and infinity.

Offload to End-plan - The duration asserted between these points is between 0 and infinity.

Capacity Analysis System

The capacity analysis system analyzes the cargo requirements and determines how much cargo can be moved on each day of the plan using the number and type of aircraft sourced for each day. When given a list of requirements, the system determines the best aircraft to move each type of cargo. The aircraft selected is

used to determine the times asserted into the temporal network as described in the preceding section. The capacity analysis system starts each day with a list of requirements sorted by cargo type as well as origin and destination. A list of the aircraft available for that day is also accessible to the system. The requirements for each day are assigned to the aircraft designated in the planset as available for that day. The cargo is divided into four categories: bulk, oversize, outsize, and passenger. Each type of cargo is considered separately and each is allotted aircraft from the total number of aircraft for that day. If two C-5s and two 747s are sourced for day 1, all four aircraft are considered when moving the bulk cargo and all four are considered when moving the passenger cargo as well as the oversize and outsize cargo. By not forcing attrition of the sourced aircraft after moving one type of cargo, a better-than-best-case scenario is obtained.

After getting a list of all of one cargo type to be moved on a day, the largest tonnage is considered first. Selecting the largest cargo first may not result in the optimum efficiency in the final plan. There are other methods which can be used to assign cargo to the aircraft, but only this method was used. The largest requirement is assigned to the aircraft with the largest capacity which is greater than the cargo tonnage. If 50 tons of bulk cargo are to be moved, and there are aircraft available with 55 tons capacity and 75 tons capacity for bulk cargo, the aircraft with 55 tons capacity will be selected. If the requirement were for 65 tons, the other aircraft would be selected. If no single aircraft can carry the amount of cargo, the largest capacity aircraft transports a full load and the rest is added back into the requirements list for that day. The new list is then sorted in order of largest tonnage. If 85 tons of bulk cargo were to be moved and the previous aircraft were available, 75 tons would be moved and 10 tons would be added back into the requirements which would then be sorted to consider the largest remaining requirement. The aircraft selected is subtracted from the list of aircraft available on that day and the process is repeated. After all cargo is moved or no aircraft remain, the remaining cargo, if any, is added

to the list of cargo to be moved on the next day and the process begins again with these requirements and a new list of aircraft available on the new day.

The aircraft available on each day are considered to be available at any desired location at the beginning of that day. The list of available aircraft comes directly from the staging list developed by the MACPLAN planner. It is not realistic to allow the aircraft to be anywhere, but this provides a method of determining if sufficient airlift capability is provided on a daily basis.

The capacity analysis system calculates a daily backlog of unmoved cargo. This backlog is based on the difference between the required amount of cargo to be moved and the airlift capacity for that day. Several methods for calculating the airlift capacity were developed and tested. The first method involved allocating aircraft to move requirements based on a cumulative list of requirements for each day. The second involved only the requirements which had not been moved plus the requirements for each day.

Using a cumulative list of requirements for each day regardless of what was already moved allows a best-possible airlift capability for each day with the given requirements. If a requirement is moved on one day, it is still considered available on the next day. By adopting this convention, we avoid capacity assessments based on assumptions of previous allocation decisions. A cumulative requirements list allows the cargo-aircraft matching algorithm to select from any of the requirements up to that day to make the most efficient use of the aircraft available.

Using only the requirements for one day plus the unmoved cargo from previous days allows the cargo-aircraft matching algorithm to choose from a reduced version of the cumulative cargo list used in the previous section. Having fewer choices means that more waste may be present in the final solution. However, this method allows the system to track each requirement individually and when all of a certain requirement has been moved, it is asserted into the temporal network.

V. RESULTS

Status

The airlift plan analysis system was not completed to the point expected when the project began. The temporal reasoning system is completely built and tested and has no known bugs or problems. However, the capacity analysis system, although it provides some useful data, did not reach the level of completion proposed at the beginning of the thesis. The existing system by itself would be of minimal practical benefit to an airlift planner.

Some of the difficulties encountered were the complexity of MACPLAN and implementing the proposed analysis techniques. MACPLAN is a very large program with many hidden assumptions underlying the actual code. Many of these assumptions were not well-documented and proved extremely elusive when MACPLAN was examined in depth. Several of the initial designs for the capacity analysis system provided erroneous or incomplete information, and the UTE rate for the aircraft was extremely difficult to understand based on the written code. Many days were spent tracking down subroutines, often discovering that more functions needed to be traced to find the desired information. Some of the original ideas proposed to solve the problem at hand were extremely difficult to implement. The capacity analysis system was first conceived as an aggregate event analysis. The times required to move a certain amount of cargo would be estimated based on the aggregate capability of the resources provided. A satisfactory method of calculating this aggregate capability was not found.

Although the planning analysis system does not operate to the level envisioned, the system as described in the preceding chapters is completely functional. The temporal reasoning system functions as predicted at the start of the thesis. It is a general-purpose temporal reasoner and can be interfaced to other applications. This

system provides a very good beginning point for future research in the temporal reasoning area. Future work will only require the addition of an application-specific interface to assert the desired information into a temporal network.

The capacity analysis system provides a framework for extracting temporal constraints from a given planset. Coupled with the interface to the temporal reasoning system, the capacity analysis system provides a very rudimentary airlift capability analysis. Some areas of this system may be improved. These areas include the cargo-to-aircraft matching algorithm, the aircraft availability for each day, and aircraft UTE rate considerations.

Results of Analysis

MACPLAN does not provide time estimates for the time required to move each requirement. Only a backlog estimator is provided. The plan completion date can be estimated from MACPLAN by finding the date when there is no backlog. A sample backlog estimate provided by MACPLAN is shown in Figure 5.1. After asserting the times required to move all requirements into a temporal network, the time estimated for plan completion was found to be very close to the last date that any requirement was available. The intervals provided by the system and the requirements used to create them are shown in Appendix B. The requirements are moved close to their available date because each is moved as soon as an aircraft is available to move it and a larger requirement is not moved first. A delivery time of several days was found for some requirements which became available on particularly busy days when many requirements became available at once. Parts of each requirement were moved on the days they became available, but smaller unmoved parts were pushed to the bottom of the cargo list and not moved until there was a period of time with no new requirements and when aircraft were still available.

Comparing the backlog estimates provided by the analysis system and MACPLAN provided some interesting information. There were two methods used to

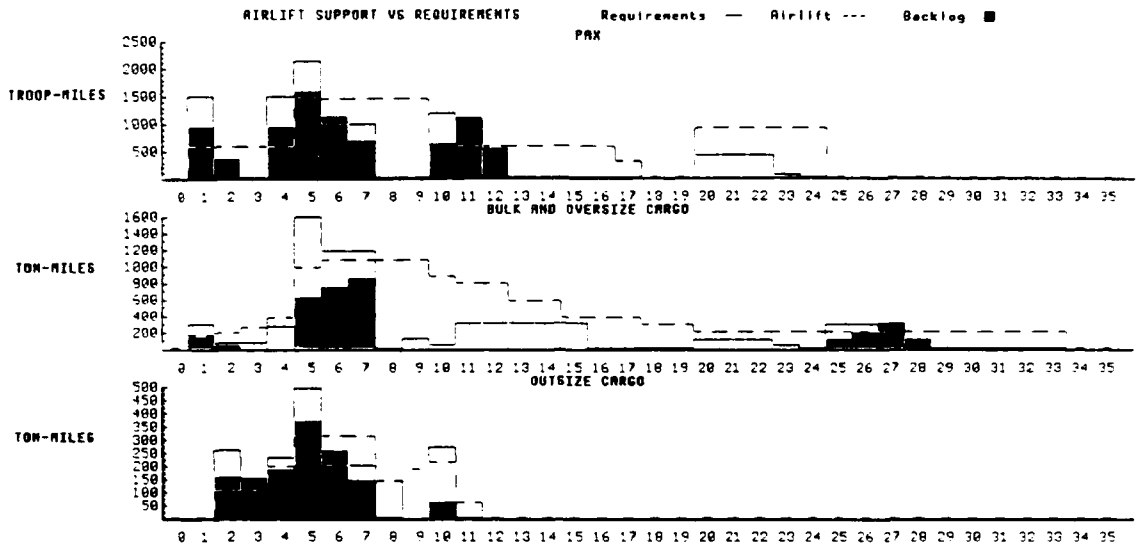

Figure 5.1. MACPLAN Backlog Estimate

calculate the daily airlift capacity in the analysis system. One involves the cumulative requirement list and the second uses each day's requirements plus unmoved cargo from previous days. Both methods provide a larger backlog estimate than the one given by MACPLAN. The cumulative requirement list provides a smaller backlog of unmoved cargo than the present day plus unmoved cargo method.

The MACPLAN backlog estimate is lower because it uses the capacity of each aircraft available on a given day to calculate the airlift capacity. This capacity is best case and is accurate only if all aircraft are completely full for each flight. In realistic plans, the cargo required to be moved does not come in blocks matching the size of the available aircraft. This causes some aircraft to make flights with less than full loads and wastes some of the airlift capability.

Both of the methods used in the capacity analysis system allocate individual cargo requirements to individual aircraft with the airlift capacity for each day computed only from the cargo moved. The difference in backlog is due to the fact that the airlift capacity is computed based on the cargo moved instead of the capacity

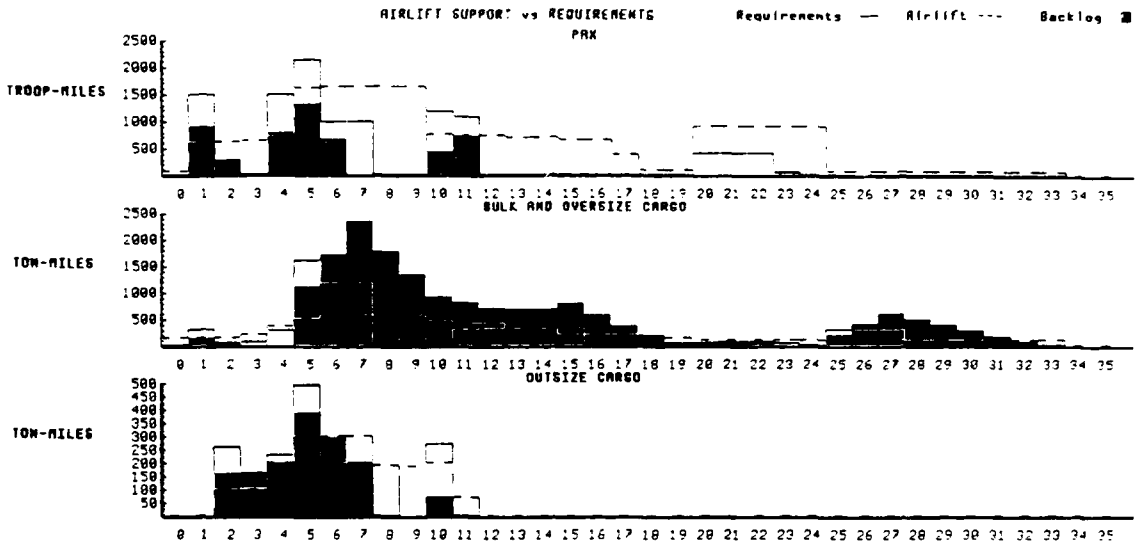

Figure 5.2. Backlog Estimate Using Cumulative Method

of the aircraft used to move it. For example, if a C-5 is used to transport 10 tons of cargo, the airlift capacity is computed as the normalized ton-miles for 10 tons instead of 47 tons which is the capacity of a C-5.

The method using the cumulative requirement list provides a smaller backlog than the method using each day's requirements plus unmoved cargo from previous days. This is because of the more efficient allocation of cargo to aircraft possible with more cargo to choose from. With more cargo to choose from, less space is wasted by using large aircraft to transport small cargo. Sample backlog graphs calculated using the two methods are shown in Figures ref backlog1 and ref backlog2. These graphs use the same requirements and planset as the MACPLAN backlog graph in Figure ref backlog.

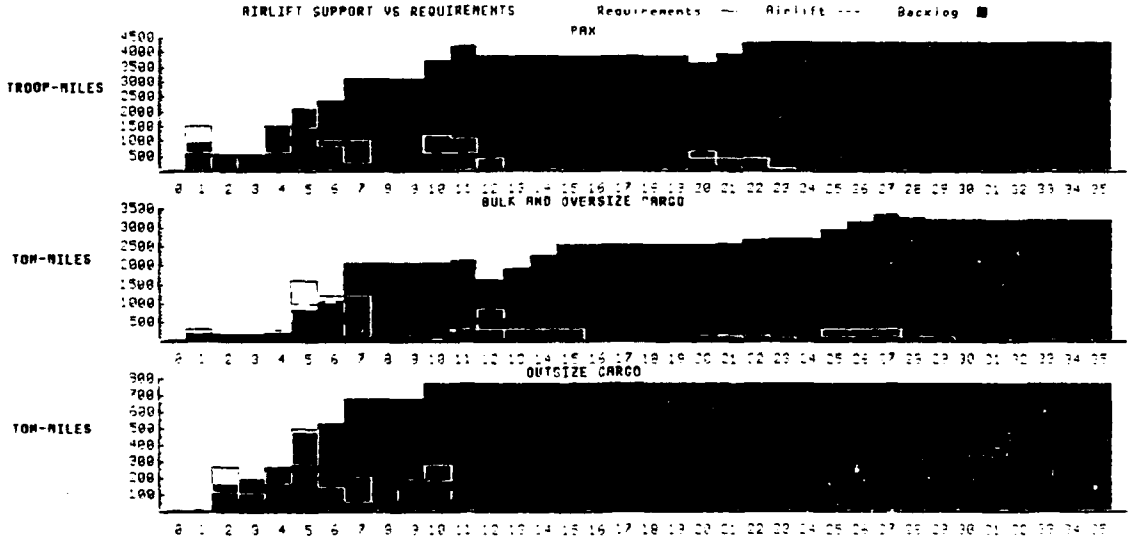

Figure 5.3. Backlog Estimate Using Unmoved Plus Method

VI. RECOMMENDATIONS

Limitations of Current System

The current airlift planning analysis system provides limited data to an airlift planner. There are several limitations in both the temporal reasoning system and the capacity analysis system. The temporal reasoning system is not capable of retracting information once the information is asserted. This can cause problems when faced with rapidly changing information or when a planner wishes to see the effects of a change in the planset. The capacity analysis system has several properties which may result in a less-than-optimal solution. These properties include the cargo-aircraft matching algorithm, the aircraft availability calculations, and the UTE rate considerations for the aircraft used.

Temporal Reasoning System Limitations. The temporal reasoning system is restricted by information that has been asserted. Once information is asserted into the system, it cannot be changed except by adding new constraints. This allows many different types of information to be represented in the same manner (temporal constraints), but does not allow for changing information. Information from separate sources which constrain the same events will be asserted correctly, i.e., the one that is the most restrictive will constrain the events. However, if the information changes at a later time, the system can not erase the constraint and replace it with another less restrictive, but still valid constraint.

The capacity analysis system does not take advantage of the disjunctive reasoning capability of the temporal reasoning system. Algorithms to assert disjunctions into the temporal network based on different choices within a planset (such as choosing a different aircraft) would utilize this capability. It would also be beneficial to track the choices which result in the different times to aid the planner in selecting the most efficient solution.

Capacity Analysis System Limitations. The capacity analysis system does not always provide the optimal matching of cargo to aircraft. The system is limited by the method used to select an aircraft to transport a requirement. The largest cargo is considered first and is matched to the largest aircraft capable of carrying it. This will provide one solution, but may not provide the most efficient solution to the problem. The priorities of the requirements and the "lateness" of each requirement are not considered in matching cargo to aircraft.

The UTE rate of the aircraft are not properly used to limit the number of hours the plane may operate over the entire plan. Each plane will be limited to trips within the UTE hour range of the aircraft selected for each day. However, the next day the aircraft will be considered available at any desired location, regardless of the final location of the aircraft on the previous day. Not tracking aircraft usage over longer periods of time may cause the proposed solution to be infeasible in the real world or less efficient than actually possible.

Recommended Future Enhancements and Research

Several enhancements to the airlift analysis system are possible. A truth maintenance system (TMS) can be added to the temporal reasoning system to properly maintain and update the system with changing information. New algorithms can be used in the capacity analysis system to improve the performance of the system. These improvements are discussed below along with future areas for further research in applying temporal reasoning to planning and scheduling problems.

Temporal Reasoning System Enhancements. The temporal reasoning system, although robust in its current state, could be improved through the addition of a truth maintenance system (TMS). A TMS would allow individual events to be tracked and when the information concerning an event changes, the latest information could be used to update the temporal network. Currently, new information will

constrain the network only if the new bounds are more restrictive than the old ones. If the old information is not valid anymore, and the new bound is less restrictive than the old one, the network must be reset to reflect these changes.

The clustering scheme used is based solely on the designator of each requirement and the type of cargo. Perhaps a better method could be found to cluster the time-points into reference sets to provide a more efficient algorithm.

Capacity Analysis System Enhancements. The cargo-to-aircraft-matching algorithm in the capacity analysis system currently matches the largest cargo to the largest aircraft available. This may result in a less-than-optimal pairing of cargo and aircraft under certain conditions. Other algorithms could be used to determine which aircraft will carry specific cargo or even algorithms not based on matching aircraft and cargo.

The cargo list is currently sorted in the order of largest to smallest when being matched to airplanes. This order could be changed to consider the priority of the cargo or the latest arrival date instead of only tonnage (or number of passengers).

The aircraft availability, or staging, is assumed to be exactly what the planner decided on in the planset. This does not account for the time required for the aircraft to fly back to the next cargo pickup point. may not provide an accurate account of the location of the planes at the end of each day. If a plane is used to fly from the US to Europe on one day, it would not be available to make the same flight the next day. However, the current system assumes the number and type of aircraft specified in the planset are available at the beginning of each day at any location needed. A method of tracking the time required to return from delivering cargo and asserting this time into the network could be implemented.

The UTE rate of each aircraft determines how many hours, on the average, the plane is available to carry cargo on each day. This number is computed by tracking each fleet of a type of aircraft and averaging the down time for all aircraft over long

periods of time. Deciding how to incorporate this into the temporal network was very difficult. An improved method of incorporating the UTE rate into the capacity analysis system to determine how much cargo an aircraft can carry over the entire plan would improve the system.

Future Research Areas. Future areas of research for applying temporal reasoning systems to planning and scheduling include improving the current system as developed so far, extending the requirements/resources analysis to discover other characteristics of high-level plans, and examining the issues involved in determining the optimum methods for partitioning the events into reference sets.

This thesis focused on the analysis of a planset instead of a developed schedule. The differences between analyzing a planset and a schedule are difficult to overcome. Deciding what constitutes a good schedule is easier than finding a good planset. A schedule either meets the stated requirements (all cargo delivered on time) or it has certain violations of the requirements which require easily found solutions. Analyzing a planset to determine if it will meet the requirements stated is more difficult. Characteristics of the planset will influence the degree to which it can meet the stated objectives. Finding the characteristics which most influence the ability of a planset to meet its objectives is the key to providing a useful analysis of the planset.

This thesis relied on the capacity analysis system to find the desired characteristics of a planset. Although not complete, the airlift planning analysis system provides some useful information about the time at which a plan may be completed. Other characteristics of the planset may be more useful in determining the "goodness" of a planset and methods to find and assert them into a temporal network may be useful. With the current temporal reasoning system, future research can focus on finding the characteristics of a planset that are good indicators of its effectiveness.

The size of the reference sets greatly influences the computation time required

to propagate a constraint through the network. The time required to assert a complete network increased dramatically with an increase in reference set size. While no research was accomplished in this area, future work to find the relationship between reference set size and computation time would be beneficial.

Conclusion

Although a practical planset analysis tool was not developed in this thesis, much useful work was accomplished. The temporal reasoning system is very robust and could be useful for other applications. The interface between the temporal reasoning system and the capacity analysis system provides the skeleton for a useful temporal airlift analysis tool. By changing the existing routines or adding new ones into the capacity analysis system, additional constraints can be easily asserted into the temporal network. The capacity analysis system provides several alternatives to the methods used in MACPLAN for calculating daily backlogs and many useful routines were developed to extract temporal constraints from a planset.

The results obtained from the completed research indicates that a useful high-level analysis of airlift plans is possible using temporal constraint propagation. This thesis studied only a small segment of the large planning analysis domain and further research in this area could provide significant advancements in the area of high-level airlift planning analysis.

Appendix A. *Source Code*

The following sections contain the source code for this thesis. The code was written in COMMON LISP on a Symbolics 3600 computer. Some calls are made to functions written in MACPLAN, which are not included. Several defined functions are not used in the present system. However, these functions were left in the source code because they may be useful to future work in this area.

Temporal Reasoning System

```
;;; -*- Syntax: Common-lisp; Package: MAC -*-

;  This is an implementation of the temporal constraint network software
;  implemented in Common Lisp.  This code was written by Jeff Clay.

;  A time-point is a structure with a list of durations in which the
;  point is in.  It also has a list of in-durations which are indirect
;  durations the point is in.  This structure corresponds to the nodes
;  in a temporal graph.

;  IMPORTANT - The time-point in each reference set that is in the
;  master reference set must be the earliest time-point in that
;  reference set.  This is essential to finding the interval constraints
;  between two time-points in different reference sets.

(cl:defstruct (time-point (:PRINT-FUNCTION PRINT-TIME-POINT))
  name
  durations
  in-durations
  reference-set)

;  Print-Time-Point will print the name of the time-point

(defun PRINT-TIME-POINT (time-point stream ignore)
  (format stream "<~a>" (time-point-name time-point)))

;  A duration is a structure with a beginning and ending time-point
;  and a bound.  This structure corresponds to the arcs in a
;  temporal graph.

(cl:defstruct (duration (:PRINT-FUNCTION PRINT-DURATION))
  point1
  point2
  bound)

;  Print-Duration prints a duration with the two time-points and the bound

(defun PRINT-DURATION (duration stream ignore)
  (format stream "<dur ~a ~a: ~d>" (duration-point1 duration)
                                   (duration-point2 duration)
   (duration-bound duration)))

;  Reset-durations will reset all durations in the list passed to it

(defun reset-durations (duration-list)
  (if (null duration-list)
      nil
      (progn
 (setf (duration-point1 (car duration-list)) nil)
 (setf (duration-point2 (car duration-list)) nil)
```

```
 (setf (duration-bound (car duration-list)) nil)
 (reset-durations (cdr duration-list)))))

;  Add-Reference-Set will add a reference set to the reference sets of
;  the time-point

(defun add-reference-set (some-time-point ref-set)
  (setf (time-point-reference-set some-time-point)
 (cons ref-set (time-point-reference-set some-time-point))))

;  Create-Time-Point will create a new time-point with the name given
;  and the reference set given.  The reference set must be a list.

(defun create-time-point (name ref-set)
  (set name (make-time-point :name name :reference-set ref-set)))

;  Reset-time-point will reset a time-point to have nil durations

(defun reset-time-point (time-point)
  (reset-durations (time-point-durations time-point))
  (reset-durations (time-point-in-durations time-point))
  (setf (time-point-durations time-point) nil)
  (setf (time-point-in-durations time-point) nil)
  (setf (time-point-reference-set time-point) nil))

;  *Master-Ref* is the identifier used for the master reference set.  The
;  master reference set is the reference set which contains one point from
;  all other reference sets.

(defvar *master-ref* '(0 1 2 3))

;  Neighbor-Points returns all points which are connected to a certain
;  time-point through a duration either as ending points or beginning points.

(defun neighbor-points (time-point)
  (union (mapcar #'duration-point2 (time-point-durations time-point))
  (mapcar #'duration-point1 (time-point-in-durations time-point))))

;  Add-duration adds a duration to the list of durations that
;  a time-point is in

(defun add-duration (some-time-point new-duration)
  (setf (time-point-durations some-time-point)
 (cons new-duration (time-point-durations some-time-point))))

;  Add-in-duration adds a duration to the list of in-durations that
;  a time point is in

(defun add-in-duration (some-time-point new-in-duration)
  (setf (time-point-in-durations some-time-point)
 (cons new-in-duration (time-point-in-durations some-time-point))))
```

```
;  Assert-bound will replace the bound on a duration if the new bound
;  is less than the original bound.  If the new bound creates a negative
;  cycle, an error is created and no changes are made.  A negative
;  cycle indicates an inconsistency in the database.

(defun assert-bound (some-duration new-bound)
  (let ((current-bounds (duration-bound some-duration))
 (reverse-dur (reverse-duration some-duration)))
    (cond ((and reverse-dur
  (bound<= (bound+ (car (last (duration-bound reverse-dur))) new-bound) 0))
    (error "Negative Cycle"))
   ((bound< new-bound (car (last current-bounds)))
    (setf (duration-bound some-duration)
   (if (bound-included current-bounds (reverse-duration-bound some-duration)
         new-bound)
       (insert-upper-bound current-bounds new-bound)
       (right-truncate-bound-list current-bounds new-bound)))
    (if reverse-dur
        (setf (duration-bound reverse-dur)
       (left-truncate-bound-list (duration-bound reverse-dur) (- new-bound))))
    (propagate-forward (duration-point1 some-duration) new-bound
         (time-point-durations
    (duration-point2 some-duration)))
    (propagate-backward (duration-point2 some-duration) new-bound
          (time-point-in-durations
     (duration-point1 some-duration))))
   (t
    nil))))

;  Insert-Upper-Bound will replace the existing upper bound with the new bound
;  in the correct place to maintain the bounds in ascending order.

(defun insert-upper-bound (bound-list bound)
  (cond ((null bound-list)
  nil)
 ((bound< bound (car bound-list))
  (cons bound (cdr bound-list)))
 (t (cons (car bound-list)
   (insert-upper-bound (cdr bound-list) bound)))))

;  Left-Truncate-Bound-List will truncate any bounds less than the new
;  bound in the bound list

(defun left-truncate-bound-list (bound-list bound)
  "truncate bounds less than bound"
  (cond ((null bound-list) nil)
 ((bound< (car bound-list) bound)
  (left-truncate-bound-list (cdr bound-list) bound))
 (t bound-list)))
```

```
;  Right-Truncate-Bound-List will truncate any bounds greater than
;  the new bound in the bound list

(defun right-truncate-bound-list (bound-list bound)
  "truncate bounds greater than bound"
  (cond ((null bound-list) nil)
 ((bound< (car bound-list) bound)
  (cons (car bound-list) (right-truncate-bound-list (cdr bound-list) bound)))
 (t nil)))

;  Bound-Included returns T if the bound given to it is between any of
;  the intervals made by the current-bounds and reverse-bounds.
;  Otherwise, it returns nil.

(defun bound-included (current-bounds reverse-bounds bound)
  (bound-included-aux current-bounds (reverse reverse-bounds) bound))

(defun bound-included-aux (current-bounds reverse-bounds bound)
  (cond ((or (null bound)
      (null current-bounds))
  nil)
 ((and (bound< bound (car current-bounds))
       (bound< (- bound) (car reverse-bounds)))
  t)
 (t (bound-included-aux (cdr current-bounds)
          (cdr reverse-bounds)
          bound))))

;  Reverse-Duration returns the duration which is the reverse
;  of the duration given it as an argument.

(defun reverse-duration (some-duration)
  (let ((point1 (duration-point1 some-duration))
 (point2 (duration-point2 some-duration)))
    (if (duration-to point2 point1)
 (duration-to point2 point1)
 nil)))

;  Reverse-Duration-Bound returns the bound-list of the reverse
;  duration of the duration given it.

(defun reverse-duration-bound (some-duration)
  (let ((reverse-dur (reverse-duration some-duration)))
    (if reverse-dur
        (duration-bound reverse-dur)
 (list nil))))

;  Bound< returns T if the first bound is less than the second,
;  otherwise, it returns nil.  Null-negative? controls whether null
;  bounds are interpreted as infinity or negative infinity.
```

```
(defun bound< (bound1 bound2 &optional (null-negative? nil))
  (if null-negative?
      (or (and (null bound1) bound2)
   (and (numberp bound2) (< bound1 bound2)))
      (or (and (null bound2) bound1)
   (and (numberp bound1) (< bound1 bound2)))))

;  Bound<= returns t if the first bound is less than or equal to
;  the second bound.  Otherwise, it returns nil.  Null-negative?
;  controls whether null bounds are interpreted as infinity or
;  negative infinity.

(defun bound<= (bound1 bound2 &optional (null-negative? nil))
  (not (bound< bound2 bound1 null-negative?)))

;  Bound+ returns the sum of two bounds.  If one of the bounds is
;  nil (or infinity), the result is nil.

(defun bound+ (bound1 bound2)
  (if (and (numberp bound1) (numberp bound2))
      (+ bound1 bound2)
      nil))

;  Propagate-forward will propagate a new bound forward through the
;  network until it has changed all affected bounds.

(defun propagate-forward (some-time-point new-bound duration-list)
  (cond ((null duration-list)
  nil)
 ((same-reference-set? some-time-point (duration-point2 (car duration-list)))
  (assert-duration some-time-point (duration-point2 (car duration-list))
     (bound+ new-bound
      (car (last (duration-bound (car duration-list))))))
  (propagate-forward some-time-point new-bound (cdr duration-list)))
 (t
  (propagate-forward some-time-point new-bound (cdr duration-list)))))

;  Propagate-backward will propagate a new bound backward through the
;  network until it has changed all affected bounds.

(defun propagate-backward (some-time-point new-bound duration-list)
  (cond ((null duration-list)
  nil)
 ((same-reference-set? some-time-point (duration-point1 (car duration-list)))
  (assert-duration (duration-point1 (car duration-list)) some-time-point
     (bound+ new-bound
      (car (last (duration-bound (car duration-list))))))
  (propagate-backward some-time-point new-bound (cdr duration-list)))
 (t
  (propagate-backward some-time-point new-bound (cdr duration-list)))))
```

```
;  Duration-to calls find-other-point with the list of durations
;  associated with time-point1 and time-point2.  Find-other-point will
;  return the duration from time-point1 to time-point2.

(defun duration-to (time-point1 time-point2)
  (if (or (null time-point1)
   (null time-point2))
      nil
      (find-other-point (time-point-durations time-point1) time-point2)))

;  Find-other-point returns the duration that is in the list of
;  durations given that time-point2 is in if it is in the list

(defun find-other-point (durations time-point2)
  (cond ((null durations) nil)
 ((eq time-point2 (duration-point2 (car durations)))
  (car durations))
 (t (find-other-point (cdr durations) time-point2))))

;  Assert-duration will assert a duration from one time-point to
;  another with the bound given it.

(defun assert-duration (point-1 point-2 bound)
  (cond ((equal point-1 point-2)
  nil)
 (t
  (let ((current-duration (duration-to point-1 point-2)))
    (if current-duration
        (assert-bound current-duration bound)
        (let ((new-duration (make-duration :point1 point-1
        :point2 point-2
        :bound '(nil))))
   (add-duration point-1 new-duration)
   (add-in-duration point-2 new-duration)
   (assert-bound new-duration bound)
   (propagate-forward point-1 bound
        (time-point-durations point-2))
   (propagate-backward point-2 bound
         (time-point-in-durations point-1)))))))


;  Direct-Duration-Bound returns the bounds from one time-point
;  to another.

(defun direct-duration-bound (one-point other-point)
  (if (or (null one-point)
   (null other-point))
      (list (list nil) (list nil))
      (let ((direct-duration (duration-to one-point other-point)))
 (if direct-duration
     (list (duration-bound direct-duration)
```

```
    (reverse-duration-bound direct-duration))
     (connection one-point other-point)))))

;  Interval-Constraint returns the intervals in which the second time-
;  point can follow the first time-point.

(defun interval-constraint (point1 point2)
  (let ((bounds (direct-duration-bound point1 point2)))
    (translate-bounds (car bounds)
        (reverse (cadr bounds)))))

;  Translate-Bounds will translate the bounds into intervals.

(defun translate-bounds (bounds-1 bounds-2)
  (cond ((or (null bounds-1)
      (null bounds-2))
  nil)
 (t
  (cons (interpret-bounds (car bounds-1)
     (car bounds-2))
        (translate-bounds (cdr bounds-1)
     (cdr bounds-2))))))

;  Interpret-Bounds interprets null bounds as infinity and
;  numbers as numeric bounds for printing out the intervals.

(defun interpret-bounds (bound rev-bound)
  (list (if (numberp rev-bound)
     (- rev-bound)
     nil)
 (if (numberp bound)
     bound
     nil)))

;  Same-Reference-Set? returns t if the two time-points are in
;  the same reference set.  Otherwise it returns nil.

(defun same-reference-set? (point1 point2)
  (common-element (time-point-reference-set point1)
    (time-point-reference-set point2)))

;  Common-element returns t if any member of one of the sets
;  is also in the other set.

(defun common-element (set1 set2)
  (cond ((null set1)
  nil)
 ((member (car set1) set2)
  t)
 (t (common-element (cdr set1) set2))))
```

```
;  Connection returns the intervals between two time-points that
;  are not in the same reference set.

(defun connection (point1 point2)
  (let ((ints (sum-interval-constraint (dur-to-master point1)
    (sum-interval-constraint (master-connect point1 point2)
        (dur-from-master point2)))))
    (list (extract-upper-bound ints)
   (extract-lower-bound ints))))

;  Dur-to-Master returns the intervals between the two time-points
;  which connect the reference set containing the provided time-point
;  to the master reference-set.  If the provided time-point is in the
;  master reference set, an interval of (0 0) is returned.

(defun dur-to-master (point1)
  (if (in-reference-set? point1 *master-ref*)
      '((0 0))
      (interval-constraint point1
      (connection-point (time-point-durations point1)
          *master-ref*
          (time-point-reference-set point1)))))

;  Master-Connect returns the interval between the two time-points in
;  the master reference set which connect the two reference sets containing
;  the given time-points.

(defun master-connect (point1 point2)
  (cond ((in-reference-set? point1 *master-ref*)
  (interval-constraint point1
         (connection-point (time-point-durations point2)
      *master-ref*
      (time-point-reference-set point2))))
 ((in-reference-set? point2 *master-ref*)
  (interval-constraint (connection-point (time-point-durations point1)
      *master-ref*
      (time-point-reference-set point1))
         point2))
 (t
  (interval-constraint (connection-point (time-point-durations point1)
      *master-ref*
      (time-point-reference-set point1))
         (connection-point (time-point-durations point2)
      *master-ref*
      (time-point-reference-set point2))))))

;  Dur-From-Master will return the intervals between the two time-points
;  which connect the master reference set to the reference set containing
;  the provided time-point.  If the provided time-point is in the master
;  reference set, an interval of (0 0) is returned.
```

```
(defun dur-from-master (point2)
  (if (in-reference-set? point2 *master-ref*)
      '((0 0))
      (interval-constraint (connection-point (time-point-durations point2)
          *master-ref*
          (time-point-reference-set point2))
        point2)))

;  Connection-Point returns the time-point from the duration-list
;  which is contained in the reference set provided.

(defun connection-point (duration-list ref-set1 ref-set2)
  (if duration-list
      (let ((a-duration (car duration-list)))
 (if (and (in-reference-set? (duration-point2 a-duration) ref-set1)
   (in-reference-set? (duration-point2 a-duration) ref-set2))
     (duration-point2 a-duration)
     (connection-point (cdr duration-list) ref-set1 ref-set2)))))

;  In-reference-set? returns T if the time point is in the reference set

(defun in-reference-set? (some-time-point reference-list)
  (if reference-list
      (if (member (car reference-list)
    (time-point-reference-set some-time-point))
   t
   (in-reference-set? some-time-point (cdr reference-list)))))

;  Assert-interval will assert two bounds on the durations between
;  the two time-points to limit the possible time relationship between
;  the two to be within the supplied interval.

(defun assert-interval (time-point1 time-point2 lower-bound upper-bound)
  (if (null lower-bound)
      (assert-duration time-point2 time-point1 nil)
      (assert-duration time-point2 time-poin . (- lower-bound)))
  (assert-duration time-point1 time-point2 upper-bound))

;  Assert-Not-Interval will assert the appropriate bounds on durations
;  to assure the time interval provided does not contain the time-
;  points provided.

(defun assert-not-interval (time-point1 time-point2 lower-bound upper-bound)
  (assert-new-interval-constraint
    time-point1 time-point2
    (list (list nil lower-bound) (list upper-bound nil))))

;  Assert-new-interval-constraint asserts the disjunction interval if
;  it further limits the time-points durations.

(defun assert-new-interval-constraint (point1 point2 new-int-constraint)
```

```
  (let ((merged-int-constraint
   (intersect-disjoint-intervals
     (interval-constraint point1 point2) new-int-constraint)))
    (if (not (equal (interval-constraint point1 point2) merged-int-constraint))
 (if (duration-to point1 point2)
     (progn
       (setf (duration-bound (duration-to point1 point2))
      (extract-upper-bound merged-int-constraint))
       (if (duration-to point2 point1)
    (setf (duration-bound (duration-to point2 point1))
   (reverse (extract-lower-bound merged-int-constraint)))
    (progn
      (let ((new-duration (make-duration :point1 point2
             :point2 point1
             :bound '(nil))))
        (add-duration point2 new-duration)
        (add-in-duration point1 new-duration)
        (setf (duration-bound new-duration)
       (reverse (extract-lower-bound merged-int-constraint))))))
       (propagate-interval-constraint
  point1 point2 merged-int-constraint
  (cl:set-difference (union (neighbor-points point1)
           (neighbor-points point2))
    (list point1 point2))))
     (progn
       (assert-interval point1 point2 nil nil)
       (assert-new-interval-constraint point1 point2 new-int-constraint))))))

;  Propagate-Interval-Constraint will propagate the new interval to
;  all other points connected to the provided time-points

(defun propagate-interval-constraint (point1 point2
                                      new-int-constraint other-points)
  (loop for point in other-points
 do (if (same-reference-set? point point1)
        (assert-new-interval-constraint
   point1 point
   (sum-interval-constraint
     new-int-constraint (interval-constraint point2 point))))
    (if (same-reference-set? point point2)
        (assert-new-interval-constraint
   point point2
   (sum-interval-constraint
     new-int-constraint (interval-constraint point point1))))))

;  Sum-Interval-Constraint will add two interval lists together to get the
;  appropriate sum interval

(defun sum-interval-constraint (icon1 icon2)
  (cl:sort (combine-intervals (interval-cross-product icon1 icon2)) #'<
 :key #'car))
```

```
;  Interval-Cross-Product will take each interval in the first list
;  and add it to each interval in the second list

(defun interval-cross-product (int-constraint1 int-constraint2)
  (if (or (null int-constraint1)
   (null int-constraint2))
      nil
      (append (cross-first (car int-constraint1) int-constraint2)
       (interval-cross-product (cdr int-constraint1) int-constraint2))))

;  Cross-First will take one interval and add it together with each interval
;  in the interval-list

(defun cross-first (interval interval-list)
  (if (null interval-list)
      nil
      (cons (list (bound+ (car interval) (caar interval-list))
    (bound+ (cadr interval) (cadar interval-list)))
     (cross-first interval (cdr interval-list)))))

;  Combine-Intervals will take a list of intervals and reduce it to the
;  smallest list of intervals which contain all intervals in the list.

(defun combine-intervals (interval-list)
  (let ((first-interval (car interval-list))
 (rest-intervals (cdr interval-list)))
    (cond ((null rest-intervals) interval-list)
   ((no-overlap first-interval rest-intervals)
    (cons first-interval (combine-intervals rest-intervals)))
   (t
    (combine-intervals (include-interval first-interval rest-intervals))))))

;  No-Overlap returns t if the provided interval does not overlap any
;  of the intervals in the interval-list

(defun no-overlap (interval interval-list)
  (if (null interval-list)
      t
      (let ((lb1 (car interval)) (ub1 (cadr interval))
     (lb2 (caar interval-list)) (ub2 (cadar interval-list)))
 (cond ((bound< ub1 lb2)
        (no-overlap interval (cdr interval-list)))
       ((bound< ub2 lb1 t)
        (no-overlap interval (cdr interval-list)))
       (t
        nil)))))

;  Include-Interval will combine the given interval with the appropriate
;  interval with which it overlaps and return the list containing only
;  one interval for the provided interval and the overlapping interval.
```

```
(defun include-interval (interval interval-list)
  (if (null interval-list)
      nil
      (let ((lb1 (car interval)) (ub1 (cadr interval))
     (lb2 (caar interval-list)) (ub2 (cadar interval-list)))
 (cond ((and (bound<= lb1 lb2 t)
      (bound<= ub1 ub2)
      (bound<= lb2 ub1))
        (cons (list lb1 ub2) (cdr interval-list)))
       ((and (bound<= lb2 lb1 t)
      (bound<= ub2 ub1)
      (bound<= lb1 ub2))
        (cons (list lb2 ub1) (cdr interval-list)))
       ((and (bound<= lb1 lb2 t)
      (bound<= ub2 ub1))
        (cons interval (cdr interval-list)))
       ((and (bound<= lb2 lb1 t)
      (bound<= ub1 ub2))
        (cons (list lb2 ub2) (cdr interval-list)))
       (t
        (cons (car interval-list)
              (include-interval interval (cdr interval-list)))))))))

;  Intersect-Disjoint-Intervals will return the list of intervals which
;  contains the intersection of all intervals in both lists.  If two
;  intervals in the lists overlap, the new list will contain only the
;  portion of the intervals that overlap.

(defun intersect-disjoint-intervals (interval1 interval2)
  (let ((lb1 (caar interval1)) (ub1 (cadar interval1))
 (lb2 (caar interval2)) (ub2 (cadar interval2)))
    (cond ((or (null (car interval1))
        (null (car interval2)))
    nil)
   ((and lb2 (bound< ub1 lb2))
    (intersect-disjoint-intervals (cdr interval1) interval2))
   ((and lb1 (bound< ub2 lb1))
    (intersect-disjoint-intervals interval1 (cdr interval2)))
   ((and (bound<= lb1 lb2 t)
  (bound<= ub1 ub2))
    (cons (list lb2 ub1)
   (intersect-disjoint-intervals (cdr interval1) interval2)))
   ((and (bound<= lb2 lb1 t)
  (bound<= ub2 ub1))
    (cons (list lb1 ub2)
   (intersect-disjoint-intervals interval1 (cdr interval2))))
   ((and (bound< lb1 lb2)
  (bound< ub2 ub1))
    (cons (list lb2 ub2)
   (intersect-disjoint-intervals interval1 (cdr interval2))))
```

```
   (t (cons (list lb1 ub1)
   (intersect-disjoint-intervals (cdr interval1) interval2))))))

;  Extract-Upper-Bound takes the upper bounds out of an interval list
;  so the bounds can be asserted on a duration

(defun extract-upper-bound (interval)
  (cond ((null interval)
  nil)
 (t
  (cons (cadar interval) (extract-upper-bound (cdr interval))))))

;  Extract-Lower-Bound takes the lower bounds out of an interval list
;  so the bounds can be asserted on a duration

(defun extract-lower-bound (interval)
  (cond ((null interval)
  nil)
 ((null (caar interval))
  (cons nil (extract-lower-bound (cdr interval))))
 (t
  (cons (- (caar interval)) (extract-lower-bound (cdr interval))))))
```

```
;;; -*- Package: MAC; Base: 10; Mode: LISP; Syntax: Common-Lisp -*-

(defvar *cum-reqts* nil)
(defvar *moved-reqts* nil)
(defvar *staging-list* nil)
(defvar *reqts-list* nil)
(defvar *ac-ute-list* nil)
(defvar *unmoved-bulk* nil)
(defvar *unmoved-oversize* nil)
(defvar *unmoved-outsize* nil)
(defvar *unmoved-pax* nil)

;  Cum-req-list returns a list containing all loaded requirements each in the
;  form (day <onload-station> <offload-station> (bulk oversize outsize pax))

(defun cum-req-list ()
  (cl:sort (cleanup-by-day-req-list (by-day-req-list))
    #'<
    :key #'car))

;  Cumulative-reqts requires a list of the form returned by cum-req-list.
;  It returns a list of the same format, but any requirement will contain
;  the sum of all requirements on earlier days which have the same onload
;  and offload stations.  For example, if two requirements have the same
;  source and destination, and are on days 5 and 10 respectively, the
;  requirement on day 10 will contain the sum of the two requirements
;  while the one on day 5 will only contain the requirement for day 5.

(defun cumulative-reqts ()                    ; NO CALLERS
  (setf *cum-reqts* nil)
  (let ((req-list (cum-req-list)))
    (do ((i 0 (+ i 1)))
 ((> i (caar (last req-list))) (reverse *cum-reqts*))
      (push (cons i (accumulate-reqts (reqts-to-day i req-list)))
     *cum-reqts*))))

;  Reqts-to-day will return a list of all requirements which are
;  available on or before the given day.  If there is duplication
;  of onload and offload stations, they will NOT be combined in this
;  function.  Accumulate-reqts will combine the requirements with
;  the same onload and offload stations.

(defun reqts-to-day (day req-list)
  (if req-list
      (if (= (caar req-list) day)       ; use for single day
;      (if (<= (caar req-list) day)      ; use for cumulative
   (cons (cdar req-list)
  (reqts-to-day day (cdr req-list)))
;   )))                                  ;use for cumulative
   (reqts-to-day day (cdr req-list)))))      ; use for single day
```

```
;  Cum-reqts-to-day will return a list containing the day and a
;  cumulative list of all requirements up to that day with all
;  requirements with the same onload and offload stations added
;  together.

(defun cum-reqts-to-day (day)
  (accumulate-reqts (reqts-to-day day (cum-req-list))))

;  Accumulate-reqts will add all requirements in the list which have
;  the same onload and offload stations together into one requirement.

(defun accumulate-reqts (req-list)
  (if req-list
      (if (unique-stations (car req-list) (cdr req-list))
   (cons (car req-list)
  (accumulate-reqts (cdr req-list)))
   (accumulate-reqts (add-like-reqts (car req-list) (cdr req-list))))))

;  Unique-stations will return T if the given req does not have the same
;  onload and offload stations as any other requirement in the list.

(defun unique-stations (req req-list)
  (if req-list
      (if (not (and (equal (car req) (caar req-list))
      (equal (cadr req) (cadar req-list))))
   (unique-stations req (cdr req-list)))
      t))

;  Add-like-reqts will take the given requirement and search through the
;  given requirement list and find any requirement with the same onload
;  and offload stations and add the tonnages to the requirements in the
;  list.

(defun add-like-reqts (req req-list)
  (if req-list
      (if (and (equal (car req) (caar req-list))      ; same onload-stations
        (equal (cadr req) (cadar req-list)))   ; same offload stations
   (accumulate-reqts (cons (list (car req)     ; onload station
     (cadr req)    ; offload station
     (add-tonnages (caddr req)
           (cadr (cdar req-list)))
     (append (get-load-designator req)
      (get-load-designator (car req-list))))
      (cdr req-list)))
   (cons (car req-list)
  (add-like-reqts req (cdr req-list))))))

;  Add-Tonnages will add the corresponding tonnages of two lists.
;  The lists must be of the form (BULK OVERSIZE OUTSIZE PAX) where
;  BULK is the tons of bulk cargo, OVERSIZE is the tons of oversize
```

```
;  cargo, etc.  This is the format used in this program.  Add-tonnages
;  will add the first four elements of any two lists.

(defun add-tonnages (tons-1 tons-2)
  (list (+ (car tons-1) (car tons-2))
 (+ (cadr tons-1) (cadr tons-2))
 (+ (caddr tons-1) (caddr tons-2))
 (+ (car (cdddr tons-1)) (car (cdddr tons-2)))))

;  Cleanup-by-day-req-list cleans up the by-day-req-list by pulling
;  out the day of each requirement and changing the form of the
;  requirement list from (onload-station offload-station (day bulk ...))
;  to (day onload-station offload-station (bulk ...)).  This allows
;  easier sorting of the lists into chronological order.

(defun cleanup-by-day-req-list (by-day-req-list)
  (if by-day-req-list
      (cons (pull-out-day (car by-day-req-list))
    (cleanup-by-day-req-list (cdr by-day-req-list)))))

;  Pull-out-day will pull the day out of the cargo list and place
;  it in the front of the list.  The formats for the lists are
;  shown under cleanup-by-day-req-list.

(defun pull-out-day (req-list)
  (cons (car (caddr req-list))
 (append (list (car req-list)                    ; onload station
       (cadr req-list)                  ; offload station
       (cdr (caddr req-list)))          ; cargo list
  (list (get-load-designator req-list)))))  ;load designator

;  By-day-req-list returns a list of the loaded requirements each in the
;  form (onload-station offload-station (day bulk oversize outsize pax)).

(defun by-day-req-list ()
  (build-req-list (requirements)))

;  Build-req-list sends each requirement to add-req-to-list to build
;  a list of all requirements.

(defun build-req-list (reqts)
  (if reqts
      (add-req-to-list (car reqts)
      (build-req-list (cdr reqts)))))

;  Add-req-to-list extract the desired data from each requirement and
;  adds it to the list of all requirements.

(defun add-req-to-list (requirement req-list)
  (cons (list (send requirement :onload-station)
       (send requirement :offload-station)
```

```
      (cons (requirement-date requirement)
     (list-tons (send requirement :cargo)))
      (list (send requirement :load-designator)))
 req-list))

;  Requirement-date calculates the available-date for each requirement.
;  This function currently uses the earliest-arrival-time from each
;  requirement.  It should be modified to use the same algorithm used
;  by MACPLAN to calculate the available-date in the load-requirements
;  function.

(defun requirement-date (requirement)
  (relative-to-relative-day (send requirement :earliest-arrival-time)))

;  List-tons returns a list containing the tonnages for the cargo list
;  given it.  The tonnages are in the order (bulk oversize outsize pax).
;  This is the order they are stored in the data base.

(defun list-tons (cargo-list)
  (if cargo-list
      (cons (send (cdar cargo-list) :tonnage)
     (list-tons (cdr cargo-list)))))

;  Aircraft-staging-on-day will return a list of the aircraft scheduled
;  for a certain day which gives the number of aircraft which are sourced
;  for that day.  The returned list is of the form
;  ((<aircraft-1> #-aircraft-1) (<aircraft-2> #-aircraft-2) ... )

(defun aircraft-staging-on-day (day)
  (build-staging-on-day (aircraft-staging-list) day))

(defun build-staging-on-day (staging-list day)
  (if staging-list
      (cons (cons (caar staging-list)
    (build-ac-staging (cadar staging-list) day 0))
     (build-staging-on-day (cdr staging-list) day))))

(defun build-ac-staging (ac-staging-list day current-number-of-ac)
  (if ac-staging-list
      (if (<= (caar ac-staging-list) day)
   (build-ac-staging (cdr ac-staging-list) day (cdar ac-staging-list))
   current-number-of-ac)
      current-number-of-ac))

;  Number-aircraft-on-day will return the number of the type of aircraft which
;  are sourced for the day given.

(defun number-aircraft-on-day (aircraft day)
  (find-aircraft-on-day aircraft (aircraft-staging-on-day day)))

(defun find-aircraft-on-day (aircraft staging)
```

```
  (if staging
      (if (equal aircraft (caar staging))
   (cdar staging)
   (find-aircraft-on-day aircraft (cdr staging)))))

;  New-run-airlift-compare is exactly like the run-airlift-compare in MACPLAN
;  except mine calculates the airlift capacity in a different way.  This function
;  does not call airlift-compare to calculate the airlift capacity although the
;  requirements are still calculated from airlift-compare.  The airlift capacity
;  is calculated by normalizing only the requirements that can be moved on a
;  given day by the aircraft scheduled for that day.

(defun new-run-airlift-compare (begin end &aux new-data)
  (if (and *demonstration* *setup-demonstration*)
      (format *dw* "~&No demonstration setup for airlift
                    requirement comparison~%")
      (format *dw* "Gathering data for overall comparison of
                   capacity and requirements...~%")
      (let (requirements data airlift backlog)
 (setq data (cons (assoc 'requirements (airlift-compare begin end))
    (list (cons 'AIRLIFT (calc-moved-reqts begin end)))))
 (setq requirements (transform-reqts (cdr (assoc 'requirements data))))
 (setq airlift (allocate-C5-capacity (cdr (assoc 'airlift data)) requirements))
 (setq backlog (calc-airlift-backlog requirements airlift begin end))
 (setq new-data (cons (cons 'requirements requirements)
                      (list (cons 'airlift airlift) (cons 'backlog backlog)))))
      (format *dw* "Done gathering data, now displaying it...~%")
      (let (plist (destination (get-phanode-instance 'we-ov-airlift-compare)))
 (if (null (assoc destination *mac-windows*))
     (push (list destination 'airlift-requirement-comparison) *mac-windows*))

 (setq plist '((STATION-LABEL "AIRLIFT SUPPORT VS REQUIREMENTS")
        (START-TIME ,begin)
        (END-TIME ,end)
        (DATA ,@new-data)))
 (send (get-phanode-instance 'air-analysis-subgraphm) :undisplay 'user)
 (if *expose-airlift-automatically*
     (send destination :display 'user plist)
     (loop for item in (send destination :children)
                   do (send item :display 'user plist))
     (format *dw* "Air support analysis now available in WINDOWS menu
                  as AIRLIFT-REQUIREMENTS-COMPARISON.~%"))
 )))

;  Calc-moved-reqts will calculate the requirements that can be moved on each
;  day between the c-begin and c-end days.  The c-begin and c-end should be in
;  the format of 'CXXX.  The moved-requirements will be used as the airlift
;  capacity for the new-run-airlift-compare function.

(defun calc-moved-reqts (c-begin c-end)
  (setf *unmoved-bulk* nil)
```

```
  (setf *unmoved-oversize* nil)
  (setf *unmoved-outsize* nil)
  (setf *unmoved-pax* nil)
  (setf *moved-reqts* nil)
  (let ((begin (relative-to-relative-day c-begin))
 (end (relative-to-relative-day c-end)))
    (do ((i begin (+ i 1)))
 ((> i end) (consolidate-moved-reqts (reverse *moved-reqts*)))
      (push (cons i (calc-moved-reqts-for-day i))  ;(cum-reqts-to-day i)
                ;(aircraft-staging-on-day i)))
     *moved-reqts*))))

;  Calc-moved-reqts-for-day will calculate how many of the
;  requirements given it can be moved with the aircraft
;  staging given it.  It will set *staging-list* to the staging
;  given it.  The *staging-list* will be modified by the functions
;  that determine if a requirement can be moved by the aircraft
;  still available.  When an aircraft is used, it will be subtracted
;  from the *staging-list*.

(defun calc-moved-reqts-for-day (day)
  (list (calc-bulk-moved-for-day day)
 (calc-over-moved-for-day day)
 (calc-out-moved-for-day day)
 (calc-pax-moved-for-day day)))

;  Calc-bulk-moved-for-day will return the number of tons of bulk cargo out
;  of the cumulative requirements for that day which can be moved by the aircraft
;  sourced for that day.

(defun calc-bulk-moved-for-day (day)
  (setf *ac-ute-list* nil)
  (setf *staging-list* (aircraft-staging-on-day day))
  (setf *reqts-list*
 (sort-cargo-list (append *unmoved-bulk*
     (remove-empty-req (sort-req-bulk (cum-reqts-to-day day))))))
  (setf *unmoved-bulk* nil)
  (cargo-move :bulk-capacity day)
  (convert-cargo-ute *ac-ute-list* day))

;  Calc-over-moved-for-day will return the number of tons of oversize cargo out
;  of the cumulative requirements for that day which can be moved by the aircraft
;  sourced for that day.

(defun calc-over-moved-for-day (day)
  (setf *ac-ute-list* nil)
  (setf *staging-list* (aircraft-staging-on-day day))
  (setf *reqts-list*
 (sort-cargo-list (append *unmoved-oversize*
     (remove-empty-req (sort-req-over (cum-reqts-to-day day))))))
  (setf *unmoved-oversize* nil)
```

```
  (cargo-move :oversize-capacity day)
  (convert-cargo-ute *ac-ute-list* day))

;  Calc-out-moved-for-day will return the number of tons of outsize cargo out
;  of the cumulative requirements for that day which can be moved by the aircraft
;  sourced for that day.

(defun calc-out-moved-for-day (day)
  (setf *ac-ute-list* nil)
  (setf *staging-list* (aircraft-staging-on-day day))
  (setf *reqts-list*
 (sort-cargo-list (append *unmoved-outsize*
    (remove-empty-req (sort-req-out (cum-reqts-to-day day))))))
  (setf *unmoved-outsize* nil)
  (cargo-move :outsize-capacity day)
  (convert-cargo-ute *ac-ute-list* day))

;  Calc-pax-moved-for-day will return the number passengers out
;  of the cumulative requirements for that day which can be moved by the aircraft
;  sourced for that day.

(defun calc-pax-moved-for-day (day)
  (setf *ac-ute-list* nil)
  (setf *staging-list* (aircraft-staging-on-day day))
  (setf *reqts-list*
 (sort-cargo-list (append *unmoved-pax*
    (remove-empty-req (sort-req-pax (cum-reqts-to-day day))))))
  (setf *unmoved-pax* nil)
  (cargo-move :pax-capacity day)
  (convert-cargo-ute *ac-ute-list* day))

;  Cargo-move is a general purpose function to determine how much of a given
;  type of cargo can be moved with the aircraft staging in *staging-list* and
;  the requirements in *reqts-list*.  The *reqts-list* is a sorted list of
;  the cargo of the type being considered for that day.  The capacity function
;  passed as an argument is used to determine the capacity of each type of
;  aircraft for the type of cargo which is being moved.  The capacity function
;  should be one of the following:  :pax-capacity, :oversize-capacity,
;  :outsize-capacity, or :bulk-capacity.  Cargo-move is ran completely for its
;  side effects of generating the *ac-ute-list* for each type of cargo.  The
;  tons moved is then computed by convert-cargo-ute.

(defun cargo-move (capacity-function day)
  (if *reqts-list*
      (let ((onload (caar *reqts-list*))
     (offload (cadar *reqts-list*))
     (cargo-moved (move-cargo (caar *reqts-list*)          ;onload station
         (cadar *reqts-list*)          ;offload station
         (caar (cddar *reqts-list*))  ;cargo-tons
         capacity-function            ;capacity function
         day)))
```

```
(if cargo-moved         ; if some of the requirement was moved
    (+ (normalize-cargo onload offload (car cargo-moved))  ;add cargo moved to
       (progn                                              ;rest of cargo moved
  (setf *reqts-list* (if (cdr cargo-moved)                 ;remove cargo from list
    (sort-cargo-list (cons (cadr cargo-moved)
            (cdr *reqts-list*)))
    (cdr *reqts-list*)))
  (cargo-move capacity-function day))) ;try to move next cargo
    (progn    ;if unable to move cargo
      (cond ((equal capacity-function ':pax-capacity)
      (setf *unmoved-pax* (cons (car *reqts-list*)
           *unmoved-pax*)))
     ((equal capacity-function ':bulk-capacity)
      (setf *unmoved-bulk* (cons (car *reqts-list*)
     *unmoved-bulk*)))
     ((equal capacity-function ':oversize-capacity)
      (setf *unmoved-oversize* (cons (car *reqts-list*)
        *unmoved-oversize*)))
     ((equal capacity-function ':outsize-capacity)
      (setf *unmoved-outsize* (cons (car *reqts-list*)
       *unmoved-outsize*))))
  (setf *reqts-list* (cdr *reqts-list*))  ;remove requirement from list
  (cargo-move capacity-function day)))) ;try to move next cargo
    0))  ;if no requirements left, return 0 tons to add to cumulative total

;  (terpri)
;  (print "The reqts are ")
;  (princ *reqts-list*)
;  (terpri)
;  (princ (eval (get-unmoved-list capacity-function)))

;  (terpri)
;  (print "This was not moved ")
;  (princ (car *reqts-list*))
;  (terpri)
;  (print "The unmoved-bulk is now ")
;  (princ *unmoved-bulk*)
;  (terpri)
;  (print "The unmoved-over is now ")
;  (princ *unmoved-oversize*)
;  (terpri)
;  (print "The unmoved-out is now ")
;  (princ *unmoved-outsize*)
;  (terpri)
;  (print "The unmoved-pax is now ")
;  (princ *unmoved-pax*)
;  (terpri)

;  Move-cargo will determine how many tons of the type of cargo given it
;  can be moved with the aircraft sourced for that day.  If an aircraft
;  is used to move some cargo, it will be subtracted from the sourcing
```

```
;  list.  If all of the cargo is moved, the number of tons moved is
;  returned.  If only part of it is moved, both the tons moved and the
;  remaining requirement are returned so the remaining requirement can be
;  added to the list of cargo which still has to be moved.

(defun move-cargo (onload offload cargo capacity-function day)
  (let ((chosen-ac (best-ac capacity-function cargo)))
    (if chosen-ac
 (let ((ac-capacity (send chosen-ac capacity-function)))
   (cond ((>= ac-capacity cargo)
   (subtract-from-staging chosen-ac)
   (add-aircraft-ute chosen-ac onload offload (normalize-cargo onload
              offload
              cargo))
   (assert-moved-req (get-load-designator (car *reqts-list*))
       chosen-ac
       (find-cargo-type capacity-function)                                  day
       onload
       offload)
   (list cargo))
  (t
   (subtract-from-staging chosen-ac)
   (add-aircraft-ute chosen-ac onload offload (normalize-cargo onload
              offload
              ac-capacity))
   (list ac-capacity
         (list onload
        offload
        (list (- cargo ac-capacity))
        (get-load-designator (car *reqts-list*)))))))))

(defun find-cargo-type (capacity-function)
  (cond ((equal capacity-function ':pax-capacity)
  'PAX-)
 ((equal capacity-function ':bulk-capacity)
  'BULK-)
 ((equal capacity-function ':oversize-capacity)
  'OVERSIZE-)
 ((equal capacity-function ':outsize-capacity)
  'OUTSIZE-)))



;  Convert-cargo-ute will convert the amount of cargo moved to the amount
;  that can be moved based on the ute-rate of the aircraft.  If some
;  aircraft are used to move cargo and the ute-rate is exceeded, the amount
;  of cargo will be cut down by the ratio of the ute-rate divided by the
;  hours flown in transporting the cargo.

(defun convert-cargo-ute (ute-list day)
  (if ute-list
```

```
     (let ((tons (car (cddar ute-list)))
     (ute-rate (aircraft-ute-rate (caar ute-list)))
     (hours (cadar ute-list))
     (number-aircraft (number-aircraft-on-day (caar ute-list) day)))
 (+ (if (> (* ute-rate number-aircraft)
    hours)
        tons
        (* tons (/ (* ute-rate number-aircraft) hours)))
    (convert-cargo-ute (cdr ute-list) day)))
 0))

;  Add-aircraft-ute will add the number of hours required for the chosen aircraft
;  to fly round trip between the onload and offload stations to the list of
;  ute-hours used so far by each type of aircraft in this requirement.

(defun add-aircraft-ute (aircraft onload offload tons-moved)
  (setf *ac-ute-list*
 (add-ute *ac-ute-list*
   aircraft
   (* (ute-hours-one-way onload offload aircraft) 2.0)
   tons-moved)))

;  Add-ute will search through the ute-list and find the element which contains
;  the aircraft chosen and then send that element to add-ute-to-ac.  If there
;  is no element for that type of aircraft, one is added to the list.

(defun add-ute (ute-list aircraft ute-hours tons-moved)
  (if ute-list
      (if (equal aircraft (caar ute-list))
   (cons (add-ute-to-ac (car ute-list) ute-hours tons-moved)
  (cdr ute-list))
   (cons (car ute-list)
  (add-ute (cdr ute-list) aircraft ute-hours tons-moved)))
      (list (list aircraft ute-hours tons-moved))))

;  Add-ute-to-ac will add the ute-hours to the cumulative total of hours
;  flown by that type of aircraft so far in satisfying this requirement.

(defun add-ute-to-ac (ute-list ute-hours tons-moved)
  (cons (car ute-list)
 (list
   (+ (cadr ute-list) ute-hours)
   (+ (caddr ute-list) tons-moved))))

;  Best-ac will trim the aircraft list down to only those aircraft that
;  can carry the type of cargo necessary.  It will return the aircraft
;  best suited to carrying the cargo provided.

(defun best-ac (capacity-function cargo-tons)
  (choose-best-ac (get-compatible-ac capacity-function
        (remove-used-ac *staging-list*))
```

```
    capacity-function
    cargo-tons))

;  Remove-used-ac removes any aircraft that has 0 as the staging number.

(defun remove-used-ac (staging-list)
  (if staging-list
      (if (> (cdar staging-list) 0)
   (cons (car staging-list)
  (remove-used-ac (cdr staging-list)))
   (remove-used-ac (cdr staging-list)))))

;  Choose-best-ac takes the first aircraft in the staging list given it
;  and compares it to the other aircraft in the list to find the aircraft
;  best suited to carry the amount of cargo given it.  It will return the
;  aircraft with the smallest capacity of the aircraft which will carry
;  the amount of cargo provided.

(defun choose-best-ac (ac-list capacity-function cargo-tons)
  (if ac-list
      (compare-ac (caar ac-list) (cdr ac-list) capacity-function cargo-tons)))

;  Compare-ac returns the aircraft with the smallest capacity among the
;  aircraft with a capacity larger than the cargo-tons given it.  The
;  capacity function determines which type of cargo the function considers.

(defun compare-ac (best-ac ac-list capacity-function cargo-tons)
  (if best-ac
      (if ac-list
   (let ((current-capacity (send best-ac capacity-function))
  (next-capacity (send (caar ac-list) capacity-function)))
     (cond ((= current-capacity cargo-tons)
     (compare-ac best-ac
          (cdr ac-list)
          capacity-function
          cargo-tons))
    ((= next-capacity cargo-tons)
     (compare-ac (caar ac-list)
          (cdr ac-list)
          capacity-function
          cargo-tons))
    ((and (< current-capacity cargo-tons)
   (< next-capacity cargo-tons))
     (compare-ac (if (> current-capacity next-capacity)
       best-ac
       (caar ac-list))
          (cdr ac-list)
          capacity-function
          cargo-tons))
    ((and (< current-capacity cargo-tons)
   (> next-capacity cargo-tons))
```

```
    (compare-ac (caar ac-list)
         (cdr ac-list)
         capacity-function
         cargo-tons))
   ((and (> current-capacity cargo-tons)
  (< next-capacity cargo-tons))
    (compare-ac best-ac
         (cdr ac-list)
         capacity-function
         cargo-tons))
   ((and (> current-capacity cargo-tons)
  (> next-capacity cargo-tons))
    (compare-ac (if (> current-capacity next-capacity)
      (caar ac-list)
      best-ac)
         (cdr ac-list)
         capacity-function
         cargo-tons))))
    best-ac)
     (error "There are no aircraft capable of carrying that cargo in
             the force-package")))

;  Subtract-from-staging will subtract one from the number of aircraft
;  in the *staging-list*.  This will make this aircraft unavailable for
;  moving any other requirements on this particular day.  This function
;  modifies the value of the global variable *staging-list*.

(defun subtract-from-staging (aircraft)
  (let ((staging-list *staging-list*))
    (setf *staging-list* (subtract-ac aircraft staging-list))))

;  Subtract-ac searches through the staging-list given it and finds the
;  staging that corresponds to the aircraft given it.  It then sends this
;  staging to subtract-one-ac to subtract one from the number of aircraft
;  sourced for that day.

(defun subtract-ac (aircraft staging-list)
  (if staging-list
      (if (equal aircraft (caar staging-list))
   (cons (subtract-one-ac (car staging-list))
  (cdr staging-list))
   (cons (car staging-list)
  (subtract-ac aircraft (cdr staging-list))))))

;  Subtract-one-ac subtracts one from the number of aircraft sourced for
;  this day.

(defun subtract-one-ac (ac-staging)
  (cons (car ac-staging)
 (- (cdr ac-staging) 1)))
```

```
;  Get-compatible-ac will return a cut-down version of the staging-list
;  with only the aircraft capable of carrying the type of cargo in the
;  capacity-function given it.  The capacity function is of the form
;  :bulk-capacity if bulk cargo is needed.

(defun get-compatible-ac (capacity-function staging-list)
  (if staging-list
      (if (> (send (caar staging-list) capacity-function) 0)
   (cons (car staging-list)
  (get-compatible-ac capacity-function (cdr staging-list)))
   (get-compatible-ac capacity-function (cdr staging-list)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  These capacity functions are not used now
;
;  Aircraft-pax-capacity returns the passenger cargo capacity
;  of the type of aircraft given it.  It returns the normal
;  passenger capacity if it is greater than zero.  Otherwise,
;  it returns the accompanying capacity of the aircraft.
;
;
;  The system currently does not consider the accompanying capacity
;  of the aircraft.  If the aircraft has a passenger capacity of 0,
;  it will not be used to transport passengers.  The accompanying
;  capacity can be added in by using the function below which is
;  now commented out.
;
;(defun aircraft-pax-capacity (aircraft)
;  (let ((normal-capacity (send aircraft :pax-capacity))
; (accomp-capacity (send aircraft :accompanying-capacity)))
;    (if (> normal-capacity 0)
; normal-capacity
; accomp-capacity)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun aircraft-pax-capacity (aircraft)
  (send aircraft :pax-capacity))

;  Aircraft-bulk-capacity returns the bulk cargo capacity
;  of the type of aircraft given it.

(defun aircraft-bulk-capacity (aircraft)
  (send aircraft :bulk-capacity))

;  Aircraft-oversize-capacity returns the oversize cargo capacity
;  of the type of aircraft given it.

(defun aircraft-oversize-capacity (aircraft)
  (send aircraft :oversize-capacity))
```

```
;  Aircraft-outsize-capacity returns the outsize cargo capacity
;  of the type of aircraft given it.

(defun aircraft-outsize-capacity (aircraft)
  (send aircraft :outsize-capacity))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun remove-empty-req (cargo-list)
  (if cargo-list
      (if (> (get-ton-req (car cargo-list)) 0)
   (cons (car cargo-list)
  (remove-empty-req (cdr cargo-list)))
   (remove-empty-req (cdr cargo-list)))))

(defun sort-cargo-list (cargo-list)
  (cl:sort cargo-list
    #'>
    :key #'get-ton-req))

;  Sort-req-bulk will sort the requirements in the list according to the
;  amount of bulk cargo in the requirement with the largest first.  It
;  will strip out the other cargo in the cargo list and return only the
;  tons of bulk cargo.

(defun sort-req-bulk (req-list)
  (sort-cargo-list (get-bulk-req req-list)))

;  Sort-req-over will sort the requirements in the list according to the
;  amount of oversize cargo in the requirement with the largest first.

(defun sort-req-over (req-list)
  (sort-cargo-list (get-over-req req-list)))

;  Sort-req-out will sort the requirements in the list according to the
;  amount of outsize cargo in the requirement with the largest first.

(defun sort-req-out (req-list)
  (sort-cargo-list (get-out-req req-list)))

;  Sort-req-pax will sort the requirements in the list according to the
;  number of passengers in the requirement with the largest first.

(defun sort-req-pax (req-list)
  (sort-cargo-list (get-pax-req req-list)))

(defun get-ton-req (req)
  (car (caddr req)))

;  Get-bulk-req will return the tons of bulk cargo in the requirement given it.
```

```
(defun get-bulk-req (req-list)
  (if req-list
      (cons (list (caar req-list)
    (cadar req-list)
    (list (caar (cddar req-list)))
    (get-load-designator (car req-list)))   ; load designator
     (get-bulk-req (cdr req-list)))))

;  Get-over-req will return the tons of oversize cargo in the
;  requirement given it.

(defun get-over-req (req-list)
  (if req-list
      (cons (list (caar req-list)
    (cadar req-list)
    (list (cadar (cddar req-list)))
    (get-load-designator (car req-list)))   ; load designator
     (get-over-req (cdr req-list)))))

;  Get-out-req will return the tons of outsize cargo in the requirement given it.

(defun get-out-req (req-list)
  (if req-list
      (cons (list (caar req-list)
    (cadar req-list)
    (list (car (cddar (cddar req-list))))
    (get-load-designator (car req-list)))   ; load designator
     (get-out-req (cdr req-list)))))

;  Get-pax-req will return the number of passengers in the requirement given it.

(defun get-pax-req (req-list)
  (if req-list
      (cons (list (caar req-list)    ; onload station
    (cadar req-list)   ; offload station
    (list (cadr (cddar (cddar req-list))))  ; pax- req
    (get-load-designator (car req-list)))   ; load designator
     (get-pax-req (cdr req-list)))))

;  Aircraft-UTE-rate returns the ute rate for the aircraft given it.
;  It currently returns the cdar or the ute-table in the UTE-JSCP
;  object associated with the aircraft.  If the UTE-table is
;  longer than 1, the UTE rate may have to be computed based on the
;  day of the plan.  This feature is not currently supported.

(defun aircraft-ute-rate (aircraft)
  (let ((ute-table (send (send aircraft :ute-jscp) :ute-table)))
    (if (= (length ute-table) 1)
 (cdar ute-table)
 (cdar ute-table))))
```

```
;  UTE-hours-one-way returns the number of hours required to fly ONE-WAY
;  form the onload-station to the offload station with the aircraft

(defun ute-hours-one-way (onload offload aircraft)
  (/ (flight-time (select-path-time (enroute-paths onload
        offload))
    (send aircraft :tas))
     60.0))

;  Consolidate-moved-reqts will take a list of all requirements that
;  can be moved and consolidate the list into a list divided by types
;  of cargo.  This puts the list into the same format as returned by
;  the airlift compare function in MACPLAN so the MACPLAN graphing
;  functions can be used.

(defun consolidate-moved-reqts (moved-req-list)
  (list (cons 'PAX (calc-pax moved-req-list))
 (cons 'C5BULK&OVER (calc-C5bulk&over moved-req-list))
 (cons 'BULK&OVER (calc-bulk&over moved-req-list))
 (cons 'OUTSIZE (calc-outsize moved-req-list))))

;  Calc-pax consolidates all of the passenger totals in the list
;  given it.

(defun calc-pax (moved-req-list)
  (if moved-req-list
      (cons (cons (caar moved-req-list) (caddr (cddar moved-req-list)))
     (calc-pax (cdr moved-req-list)))))

;  Calc-C5bulk&over consolidates all the cargo moved by C5's.
;  Not sure how to calculate what is moved by C-5's yet.

(defun calc-C5bulk&over (moved-req-list)
  (if moved-req-list
      (cons (cons (caar moved-req-list) 0)
     (calc-C5bulk&over (cdr moved-req-list)))))

;  Calc-bulk&over consolidates all the bulk and oversize cargo in the
;  list given it.

(defun calc-bulk&over (moved-req-list)
  (if moved-req-list
      (cons (cons (caar moved-req-list) (+ (cadar moved-req-list)
        (caddr (car moved-req-list))))  ;  add bulk and oversize cargo
     (calc-bulk&over (cdr moved-req-list)))))

;  Calc-outsize consolidates all of the outsize cargo in the list
;  given it.

(defun calc-outsize (moved-req-list)
  (if moved-req-list
```

```
      (cons (cons (caar moved-req-list) (caddr (cdar moved-req-list)))
     (calc-outsize (cdr moved-req-list)))))

;  Normalize cargo will return a normalized value for the tons of cargo
;  and the distance between the onload station and offload station.

(defun normalize-cargo (onload offload tons)
  (calc-normal tons
       (distance-between-stations onload offload)))

;  Calc-normal calculates the normalized value of the cargo by
;  dividing the product of the cargo and distance by 1000.

(defun calc-normal (cargo-tons distance)
  (/ (* cargo-tons distance) 1000.0))

;  Get-Load-Designator will return the load designator in a list of the
;  form (day onload offload (cargo-list) load-designator).  If this list
;  form changes, this function can be changed without changing the rest
;  of the code.

(defun get-load-designator (req)
  (car (last req)))
```

Interface System

```
;;; -*- Package: MAC; Base: 10; Mode: LISP; Syntax: Common-Lisp -*-

(defvar begin-plan)
(defvar end-plan)
(defvar distances nil)
(defvar max-distance nil)

;  Requirements will return the requirement instances which are
;  presently loaded in the current MACPLAN environment.  If no
;  requirements are loaded, it returns nil.

(defun requirements ()
   (plan-element-instances (get-descriptor 'requirements)))

;  End-station will return a list containing the onload and offload
;  stations for a given requirement.

(defun end-station (requirement)
  (cons (send requirement :onload-station)
 (list (send requirement :offload-station))))

;  Distance will return the distance as computed by the MAC function
;  great-circle-distance between the two stations given it.  The stations
;  provided must be station instances, not just the ICAO code.

(defun distance-between-stations (station1 station2)
  (great-circle-distance (latitude-to-num (send station1 :latitude))
    (longitude-to-num (send station1 :longitude))
    (latitude-to-num (send station2 :latitude))
    (longitude-to-num (send station2 :longitude))))

;  Distance-station-to-group will return the distance between a station
;  and the station selected from the group.  The distance-between-stations
;  function is used to calculate the distance between the stations.

(defun distance-station-to-group (station group)
  (distance-between-stations station (select-station-from-group-distance group)))

;  Select-station-from-group-distance will select a station from the group
;  for the purpose of calculating the distance to the group.  Now it simply
;  selects the first member station in the group.  This should not cause a
;  problem since the distance to the group will be added to the distance from
;  the group to another station.  This should produce less error than using
;  the distance-between-groups function which calculates the maximum distance
;  between stations in two groups.

(defun select-station-from-group-distance (group)
  (car (send group :member-stations)))

;  Time-to-fly will return the time required for the given aircraft
```

```
;  to fly the distance between the two stations at the TAS (True Air
;  Speed) given in the aircraft data base.  The latitude-here and
;  latitude-there are used only for telling whether or not the location
;  provided is a station or a to-from-group.  If it is a station, the
;  latitude given by (send here :latitude) will not be a number.  If
;  the location is a group, it will return a number.  It would be
;  easier if (send location :latitude) returned the same thing for
;  both stations and groups, but that code is written in MACPLAN and
;  was not modified.

(defun time-to-fly (here there tas)
  (let ((latitude-here (send here :latitude))
 (latitude-there (send there :latitude)))
    (cond ((and (not (numberp latitude-here))
  (not (numberp latitude-there)))
    (/ (distance-between-stations here there) (/ tas 60.0)))
   ((and (numberp latitude-here)
  (numberp latitude-there))
    (/ (distance-between-groups here there) (/ tas 60.0)))
   ((and (not (numberp latitude-here))
  (numberp latitude-there))
    (/ (distance-station-to-group here there) (/ tas 60.0)))
   ((and (numberp latitude-here)
  (not (numberp latitude-there)))
    (/ (distance-station-to-group there here) (/ tas 60.0)))
   (t
    nil))))

;  Flight-time will return the time required to fly the path provided
;  at the air speed provided.  No provisions are made at this time for
;  ground time.

(defun flight-time (path tas)
  (if (cadr path)
      (+ (time-to-fly (car path) (cadr path) tas)
  (flight-time (cdr path) tas))
      0.0))

;  Cargo-types will return a list of the types of cargo in the
;  requirement given it.  The list will look like
;  (BULK OUTSIZE OVERSIZE PAX) containing only the types which
;  are in the requirement.

(defun cargo-types (requirement)
    (list-cargo-types (car (cdddr requirement))))

;  List-cargo-types will build a list of cargo-types which have
;  a tonnage of greater than zero.  Even passenger (PAX) cargo
;  is listed in tonnage even though the number is the number of
;  people.
```

```
(defun list-cargo-types (cargo-list)
  (if cargo-list
      (remove nil (list (if (> (car cargo-list) 0)
       'bulk)
   (if (> (cadr cargo-list) 0)
       'oversize)
   (if (> (caddr cargo-list) 0)
       'outsize)
   (if (> (cadr (cddr cargo-list)) 0)
       'pax)))))

;  Min-tas will return the minimum TAS (True Air Speed) for all of
;  the aircraft in the aircraft list given to it.

(defun min-tas (ac-list)
  (if ac-list
      (let ((all-min-tas (mapcar #'(lambda (x) (send x :tas)) ac-list)))
 (eval (cons 'min all-min-tas)))
      (error "There are no aircraft in this list")))

(defun fastest-speed (cargo-types ac-list)
  (min-tas (fastest-required-ac cargo-types ac-list)))

;  Fastest-required-ac returns a list containing the fastest planes
;  that can carry each type of cargo in the cargo-list.
;  There will be only one plane for each type of cargo.

(defun fastest-required-ac (cargo-types ac-list)
  (if cargo-types
      (cons (fastest-ac (compatible-ac (list (car cargo-types)) ac-list))
     (fastest-required-ac (cdr cargo-types) ac-list))
      '()))

;  Fastest-ac returns the fastest aircraft in the list

(defun fastest-ac (ac-list)
  (find-fastest (car ac-list) (cdr ac-list)))

(defun find-fastest (ac ac list)
  (if ac-list
      (if (> (send (car ac-list) :tas)
      (send ac :tas))
   (find-fastest (car ac-list) (cdr ac-list))
   (find-fastest ac (cdr ac-list)))
      ac))

;  Slowest-ac returns the slowest aircraft in the list

(defun slowest-ac (ac-list)
  (find-slowest (car ac-list) (cdr ac-list)))
```

```
(defun find-slowest (ac ac-list)
  (if ac-list
      (if (< (send (car ac-list) :tas)
      (send ac :tas))
   (find-slowest (car ac-list) (cdr ac-list))
   (find-slowest ac (cdr ac-list)))
      ac))

;  Compatible-ac will return a list of all aircraft that can carry the
;  type of cargo in the cargo list from the aircraft list given it.
;  It will return a list of all aircraft that can carry any one or the
;  types of cargo, not only the ones that can carry all of the cargo
;  types.

(defun compatible-ac (cargo ac-list)
  (if ac-list
      (if (ac-carry-cargo cargo (car ac-list))
   (cons (car ac-list)
  (compatible-ac cargo (cdr ac-list)))
   (compatible-ac cargo (cdr ac-list)))))

;  Ac-carry-cargo returns T if the aircraft given it can carry any of
;  the types of cargo in the cargo list.

(defun ac-carry-cargo (cargo-types aircraft)
  (if cargo-types
      (let ((cargo (car cargo-types)))
 (if (or (and (equal  bulk cargo)
       (> (send aircraft :bulk-capacity) 0))
  (and (equal 'outsize cargo)
       (> (send aircraft :outsize-capacity) 0))
  (and (equal 'oversize cargo)
       (> (send aircraft :oversize-capacity) 0))
  (and (equal 'pax cargo)
       (> (send aircraft :pax-capacity) 0)))
     t
     (ac-carry-cargo (cdr cargo-types) aircraft)))
      nil))

;  Min-onload-time will return the minimum onload time for all aircraft
;  in the list given to it by finding the minimum onload time for each
;  specific type of cargo and then finding the maximum of these onload
;  times.

(defun min-onload-time (cargo-types ac-list)
  (max-onload-time (min-onload-required-ac cargo-types ac-list)))

;  Max-onload-time will return the largest onload time from the
;  aircraft in the list given to it.

(defun max-onload-time (ac-list)
```

```
   (let ((all-max-onload (mapcar #'(lambda (x) (send x :onload-time))
                         ac-list)))
     (eval (cons 'max (mapcar #'(lambda (x) (time-interval-to-minutes x))
      all-max-onload)))))

;  Min-onload-required-ac returns a list of the aircraft which have
;  the minimum onload time for each type or cargo in the list.  Only
;  one aircraft will be in the list for each type of cargo.

(defun min-onload-required-ac (cargo-types ac-list)
  (if cargo-types
      (cons (min-onload-ac (compatible-ac (list (car cargo-types)) ac-list))
            (min-onload-required-ac (cdr cargo-types) ac-list))
      '()))

;  Min-onload-ac returns the aircraft with the minimum onload time
;  of all the aircraft in the list.  The aircraft object is returned,
;  not just the onload time.

(defun min-onload-ac (ac-list)
  (find-min-onload (car ac-list) (cdr ac-list)))

(defun find-min-onload (ac ac-list)
  (if ac-list
      (if (> (time-interval-to-minutes (send (car ac-list) :onload-time))
             (time-interval-to-minutes (send ac :onload-time)))
   (find-min-onload (car ac-list) (cdr ac-list))
   (find-min-onload ac (cdr ac-list)))
      ac))

;  Min-offload-time will return the minimum offload time for all aircraft
;  in the list given to it by finding the minimum offload time for each
;  specific type of cargo and then finding the maximum of these offload
;  times.

(defun min-offload-time (cargo-types ac-list)
  (max-offload-time (min-offload-required-ac cargo-types ac-list)))

;  Max-offload-time will return the largest offload time from the
;  aircraft in the list given to it.

(defun max-offload-time (ac-list)
   (let ((all-max-offload (mapcar #'(lambda (x) (send x :offload-time))
                          ac-list)))
     (eval (cons 'max (mapcar #'(lambda (x) (time-interval-to-minutes x))
      all-max-offload)))))

;  Min-offload-required-ac returns a list of the aircraft which have
;  the minimum offload time for each type or cargo in the list.  Only
;  one aircraft will be in the list for each type of cargo.
```

```
(defun min-offload-required-ac (cargo-types ac-list)
  (if cargo-types
      (cons (min-offload-ac (compatible-ac (list (car cargo-types)) ac-list))
            (min-offload-required-ac (cdr cargo-types) ac-list))
      '()))

;  Min-offload-ac returns the aircraft with the minimum offload time
;  of all the aircraft in the list.  The aircraft object is returned,
;  not just the offload time.

(defun min-offload-ac (ac-list)
  (find-min-offload (car ac-list) (cdr ac-list)))

(defun find-min-offload (ac ac-list)
  (if ac-list
      (if (> (time-interval-to-minutes (send (car ac-list) :offload-time))
             (time-interval-to-minutes (send ac :offload-time)))
   (find-min-offload (car ac-list) (cdr ac-list))
   (find-min-offload ac (cdr ac-list)))
      ac))

;  Get-ac-list-from-db will return a list of all aircraft objects which are
;  referenced in the hash table associated with (get-descriptor 'aircraft)
;  This list contains aircraft, but unless there is a planset loaded, the
;  aircraft seem to have nil properties for most slots.  I need to find a
;  way to have the hash table point to the actual objects in the database.

(defun get-ac-list-from-db ()
  (mapcar #'(lambda (x) (get-object 'aircraft x))
   (let ((ac-list nil))
        (maphash #'(lambda (key ignore)
       (push  key ac-list))
  (cadr (assoc 'type (query-map-keys
                                    (plan-element-database-mapper
     (get-descriptor 'aircraft))))))
        ac-list)))

;  Days-to-minutes-earliest will convert a calendar day (such as
;  C004) and convert it to minutes equal to the earliest part of
;  the day (12:01 A.M.)

(defun days-to-minutes-earliest (day)
  (* (- day 1) 1440))

;  Assert-network will take the current requirements and planset loaded
;  into MACPLAN and build a temporal network by asserting the time-
;  points associated with each requirement.

(defun create-network ()
  (create-time-point 'begin-plan *master-ref*)
  (create-time-point 'end-plan *master-ref*)
```

```
  (create-each (requirements)))

;  Create-each will take each requirement from the list of requirements
;  returned by the function (requirements) and create the required time
;  points for each one.

(defun create-each (requirements)
  (if requirements
      (progn
 (create-req-net (car requirements))
 (create-each (cdr requirements)))))

;  Assert-one-req will take one requirement of the form
;  (day onload offload (cargo-list) load-designator) and assert
;  it into the temporal network

(defun assert-one-req (req)
  (create-req-net req)
  (assert-requirement req))

;  Create-req-net will take each requirement and get the load-
;  designator number (such as R24) and create the required
;  time-points for each requirement.

(defun create-req-net (requirement)
  (let ((req-num (string (send requirement :load-designator))))
    (create-time-point
       (intern (make-symbol (string-append "AVAILABLE-BULK-" req-num)))
  (list 0 (intern (make-symbol (string-append "BULK-" req-num)))))
    (create-time-point
       (intern (make-symbol (string-append "ONLOAD-BULK-" req-num)))
  (list (intern (make-symbol (string-append "BULK-" req-num)))))
    (create-time-point
       (intern (make-symbol (string-append "LAUNCH-BULK-" req-num)))
 (list (intern (make-symbol (string-append "BULK-" req-num)))))
    (create-time-point
       (intern (make-symbol (string-append "LAND-BULK-" req-num)))
 (list (intern (make-symbol (string-append "BULK-" req-num)))))
    (create-time-point
       (intern (make-symbol (string-append "OFFLOAD-BULK-" req-num)))
 (list (intern (make-symbol (string-append "BULK-" req-num)))))
    (create-time-point
       (intern (make-symbol (string-append "AVAILABLE-OVERSIZE-" req-num)))
  (list 1 (intern (make-symbol (string-append "OVERSIZE-" req-num)))))
    (create-time-point
       (intern (make-symbol (string-append "ONLOAD-OVERSIZE-" req-num)))
  (list (intern (make-symbol (string-append "OVERSIZE-" req-num)))))
    (create-time-point
       (intern (make-symbol (string-append "LAUNCH-OVERSIZE-" req-num)))
 (list (intern (make-symbol (string-append "OVERSIZE-" req-num)))))
    (create-time-point
```

```
      (intern (make-symbol (string-append "LAND-OVERSIZE-" req-num)))
 (list (intern (make-symbol (string-append "OVERSIZE-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "OFFLOAD-OVERSIZE-" req-num)))
 (list (intern (make-symbol (string-append "OVERSIZE-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "AVAILABLE-OUTSIZE-" req-num)))
  (list 2 (intern (make-symbol (string-append "OUTSIZE-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "ONLOAD-OUTSIZE-" req-num)))
  (list (intern (make-symbol (string-append "OUTSIZE-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "LAUNCH-OUTSIZE-" req-num)))
 (list (intern (make-symbol (string-append "OUTSIZE-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "LAND-OUTSIZE-" req-num)))
 (list (intern (make-symbol (string-append "OUTSIZE-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "OFFLOAD-OUTSIZE-" req-num)))
 (list (intern (make-symbol (string-append "OUTSIZE-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "AVAILABLE-PAX-" req-num)))
  (list 3 (intern (make-symbol (string-append "PAX-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "ONLOAD-PAX-" req-num)))
  (list (intern (make-symbol (string-append "PAX-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "LAUNCH-PAX-" req-num)))
 (list (intern (make-symbol (string-append "PAX-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "LAND-PAX-" req-num)))
 (list (intern (make-symbol (string-append "PAX-" req-num)))))
   (create-time-point
      (intern (make-symbol (string-append "OFFLOAD-PAX-" req-num)))
 (list (intern (make-symbol (string-append "PAX-" req-num))))) ))

(defun assert-moved-req (req-num-list aircraft cargo-type day
                         onload-station offload-station)
  (if req-num-list
      (progn
 (assert-req-moved (car req-num-list)
     aircraft
     cargo-type
     day
     onload-station
     offload-station)
 (assert-moved-req (cdr req-num-list)
     aircraft
     cargo-type
     day
     onload-station
```

```
    offload-station))))

(defun assert-req-moved (req-num aircraft cargo-type day
                         onload-station offload-station)
  (let ((available (eval (read-from-string
      (string-append "AVAILABLE-"
       (string-append cargo-type req-num)))))
 (onload (eval (read-from-string
   (string-append "ONLOAD-"
          (string-append cargo-type req-num)))))
 (launch (eval (read-from-string
   (string-append "LAUNCH-"
          (string-append cargo-type req-num)))))
 (land (eval (read-from-string
        (string-append "LAND-"
         (string-append cargo-type req-num)))))
 (offload (eval (read-from-string
    (string-append "OFFLOAD-"
     (string-append cargo-type req-num))))))
    (assert-interval available onload
       (- (+ (days-to-minutes-earliest day)
      (time-interval-to-minutes (send aircraft :onload-time)))
   (caar (interval-constraint begin-plan available)))
       nil)
    (assert-interval onload launch
       0
       nil)
    (assert-interval launcn land
       (flight-time (select-path-time
        (enroute-paths onload-station
         offload-station))
      (send aircraft :tas))
       nil)
    (assert-interval land offload
       (time-interval-to-minutes (send aircra  .ffload-time))
       nil)
    (assert-interval offload end-plan
       0
       nil)))

(defun assert-time-available (requirements)
  (if requirements
      (progn
 (assert-one-available (send (car requirements) :load-designator)
        (requirement-date (car requirements)))
 (assert-time-available (cdr requirements)))))

(defun assert-one-available (req-num available-day)
  (assert-interval begin-plan
     (eval (read-from-string
      (string-append "AVAILABLE-BULK-" req-num)))
```

```
      (days-to-minutes-earliest available-day)
      nil)
   (assert-interval begin-plan
      (eval (read-from-string
       (string-append "AVAILABLE-OVERSIZE-" req-num)))
      (days-to-minutes-earliest available-day)
      nil)
   (assert-interval begin-plan
      (eval (read-from-string
       (string-append "AVAILABLE-OUTSIZE-" req-num)))
      (days-to-minutes-earliest available-day)
      nil)
   (assert-interval begin-plan
      (eval (read-from-string
       (string-append "AVAILABLE-PAX-" req-num)))
      (days-to-minutes-earliest available-day)
      nil))


;  Assert-requirement will take the requirement and assert the
;  appropriate durations between the time-points already created.

(defun assert-requirement (requirement)
  (let ((possible-ac (compatible-ac (cargo-types requirement)
        (get-ac-list-from-db)))
 (req-num (string (car (get-load-designator requirement)))))
    (let ((available (eval (read-from-string
                            (string-append "AVAILABLE-" req-num))))
   (onload (eval (read-from-string
                            (string-append "ONLOAD-" req-num))))
   (launch (eval (read-from-string
                            (string-append "LAUNCH-" req-num))))
   (land (eval (read-from-string
                            (string-append "LAND-" req-num))))
   (offload (eval (read-from-string
                            (string-append "OFFLOAD-" req-num)))) )
      (assert-interval begin-plan available
         (days-to-minutes-earliest (car requirement))
         nil)
      (assert-interval available onload
         (min-onload-time
    (cargo-types requirement)
    possible-ac)
         nil)
      (assert-interval onload launch
         0
         nil)
      (assert-interval launch land
         (flight-time (select-path-time (path-list requirement))
                      (fastest-speed
                                              (cargo-types requirement)
```

```
            possible-ac))
          nil)
      (assert-interval land offload
         (min-offload-time
    (cargo-types requirement)
    possible-ac)
         nil)
      (assert-interval offload end-plan
         0
         nil))))

;  Reset-network will reset all time-points and durations to nil

(defun reset-network ()
  (reset-time-point begin-plan)
  (reset-time-point end-plan)
  (reset-net (requirements)))

;  Reset-net resets all time-points associated with the requirements.

(defun reset-net (requirements)
  (if requirements
      (let ((req-num (string (send (car requirements) :load-designator))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "AVAILABLE-BULK-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "ONLOAD-BULK-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "LAUNCH-BULK-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "LAND-BULK-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "OFFLOAD-BULK-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "AVAILABLE-OVERSIZE-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "ONLOAD-OVERSIZE-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "LAUNCH-OVERSIZE-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "LAND-OVERSIZE-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "OFFLOAD-OVERSIZE-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "AVAILABLE-OUTSIZE-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "ONLOAD-OUTSIZE-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "LAUNCH-OUTSIZE-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "LAND-OUTSIZE-" req-num)))))
```

```
 (reset-time-point (eval (intern (make-symbol
                              (string-append "OFFLOAD-OUTSIZE-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "AVAILABLE-PAX-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "ONLOAD-PAX-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "LAUNCH-PAX-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "LAND-PAX-" req-num)))))
 (reset-time-point (eval (intern (make-symbol
                              (string-append "OFFLOAD-PAX-" req-num)))))
 (reset-net (cdr requirements)) )))

;  Groups will return all to-from-groups in the currently loaded planset

(defun groups ()
  (plan-element-instances (get-descriptor 'to-from-group)))

;  Path-list will return a list of all stations and groups in a path
,  from one station to another

(defun path-list (requirement)
  (let ((onload-station (cadr requirement))
 (offload-station (caddr requirement)))
    (enroute-paths onload-station offload-station)))

;  Enroute-paths will return a list of all enroute stations and groups
;  which are on a path from one station to another

(defun enroute-paths (onload-station offload-station)
  (let ((begin-group (parent-group onload-station (groups)))
 (end-group (parent-group offload-station (groups))))
    (list-paths onload-station offload-station
  (find-paths begin-group
       end-group
       (paths)))))

;       (send begin-group :paths)))))

;  List-paths will return a list containing the onload station, the
;  enroute groups, and the offload station for each path in the path
;  list given it.  If more than one path exists between two stations,
;  this function adds the onload and offload stations to the enroute
;  groups returned by enroute-paths to give complete paths.

(defun list-paths (onload-station offload-station path-list)
  (if path-list
      (cons (cons onload-station
    (reverse (cons offload-station
     (reverse (list-groups (cdr (send (car path-list)
```

```
          :links)))))))
      (list-paths onload-station offload-station (cdr path-list)))))

;  List-groups will return the groups which are the start-groups for each
;  link in the link list.  Notice that the first link is thrown away before
;  calling list-groups so the group of the onload-station is not included
;  in the list.

(defun list-groups (link-list)
  (if link-list
      (cons (send (car link-list) :start-group)
     (list-groups (cdr link-list)))))

;  Find-paths will find all paths (PATH objects) which connect two groups.

(defun find-paths (begin-group end-group path-list)
  (if path-list
      (if (correct-path? begin-group end-group (car path-list))
   (cons (car path-list)
  (find-paths begin-group end-group (cdr path-list)))
   (find-paths begin-group end-group (cdr path-list)))))

;  Correct-path? will return t if the path ends in the end-group.

(defun correct-path? (begin-group end-group path)
  (and (equal end-group (send path :destination))
       (equal begin-group (send path :origin))))

;  Select-path-time will return a path from the list given it according
;  to the time required to traverse that path.  The function now simply
;  returns the first path in the list given it.  If criteria are found to
;  select a path, they can be added in later.

(defun select-path-time (path-list)
  (car path-list))

;  Parent-group will return the to-from-group which contains the station.

(defun parent-group (station group-list)
  (if group-list
      (if (member station (send (car group-list) :member-stations))
   (car group-list)
   (parent-group station (cdr group-list)))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  The functions below are only for diagnostic purposes to find all
;  paths in a given planset
;
;  Tell-all-paths returns all paths which correspond to the
;  requirements.  If more than one path matches a requirement,
;  they will all be returned.
```

```
(defun tell-all-paths ()
  (list-each-path (get-all-paths (requirements))))

;  Get-all-paths will get all paths for all requirements given it.

(defun get-all-paths (requirement-list)
  (if requirement-list
      (cons (path-list (car requirement-list))
     (get-all-paths (cdr requirement-list)))))

;  List-each-path will take each path from the list and send them to
;  name-path one at a time.

(defun list-each-path (path-list)
  (if path-list
      (cons (name-path (car path-list))
     (list-each-path (cdr path-list)))))

;  Name-path will take all paths which correspond to a single requirement
;  and send them to name-each-path.

(defun name-path (path)
  (if path
      (cons (name-each-path (car path))
     (name-path (cdr path)))))

;  Name-each-path will return the names of all stations or groups which
;  are contained in the path sent to it.

(defun name-each-path (path)
  (if path
      (cons (send (car path) :name)
     (name-each-path (cdr path)))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;  List-requirements will return a list containing the parameters asked for
;  in list-one-requirement for each requirement loaded into MACPLAN.

(defun list-requirements ()
  (list-each-requirement (requirements)))

;  List-each-requirement will take each requirement separately and send
;  them to list-one-requirement.

(defun list-each-requirement (requirement-list)
  (if requirement-list
      (cons (list-one-requirement (car requirement-list))
     (list-each-requirement (cdr requirement-list)))))

;  List-one-requirement will return the desired parameters for the
; requirement given it.
```

```
(defun list-one-requirement (requirement)
  (list (send requirement :load-designator)
 (send (send requirement :onload-station) :name)
 (send (send requirement :onload-station) :icao)
 (send (send requirement :offload-station) :name)
 (send (send requirement :offload-station) :icao)
 (cargo-types requirement)))

;  Paths returns all paths in the current plan.

(defun paths ()
  (plan-element-instances (get-descriptor 'path)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  The paths functions below are used only to look at the paths in a
;  given planset
;
;  Tell-paths will return a list of the official-names of all paths
;  in the current plan.

(defun tell-paths ()
  (list-the-path (paths)))

;  List-the-path sends each path one at a time to list-a-path.

(defun list-the-path (path-list)
  (if path-list
      (cons (list-a-path (car path-list))
     (list-the-path (cdr path-list)))))

;  list-a-path will print out the official-name of the path given it.

(defun list-a-path (path)
  (princ (send path :official-name))
  (terpri))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;  Stations returns a list of all stations loaded into the system
(defun stations-list ()
  (plan-element-instances (get-descriptor 'station)))

;  Force-packages returns a list of all force-packages loaded into the system

(defun force-packages ()
  (plan-element-instances (get-descriptor 'force-package)))

;  Aircraft-staging-list will return a list of the staging for each aircraft
;  type in the loaded force-packages.  The list will contain each aircraft
;  object followed by the staging list for that aircraft such as
;  ((<aircraft1> ((day . #) (day . #))) (<aircraft2> ((day . #) (day . #))))
```

```
(defun aircraft-staging-list ()
  (build-staging-list (force-packages)))

;  Build-staging-list takes each force-package and sends it to one-staging
;  and cons the resulting lists together to form one final staging list
;  containing all force-packages.

(defun build-staging-list (force-packages)
  (if force-packages
      (cons (one-staging (car force-packages))
     (build-staging-list (cdr force-packages)))))

;  One-staging returns the aircraft object contained in the force-package in a
;  list with the staging list by days.

(defun one-staging (force-package)
  (list (send force-package :configuration)
 (send force-package :staging)))


(defun analyze-plan (begin end)
  (create-network)
  (assert-time-available (requirements))
  (new-run-airlift-compare begin end))
```

Appendix B. *Temporal Network and Requirements*

Temporal Network

```
(interval-constraint begin-plan offload-bulk-r1)
((NIL NIL))
(interval-constraint begin-plan offload-bulk-r2)
((18040.798161788025d0 NIL))
(interval-constraint begin-plan offload-bulk-r3)
((29674.442182995117d0 NIL))
(interval-constraint begin-plan offload-bulk-r4)
((NIL NIL))
(interval-constraint begin-plan offload-bulk-r5)
((29621.541821038183d0 NIL))
(interval-constraint begin-plan offload-bulk-r6)
((NIL NIL))
(interval-constraint begin-plan offload-bulk-r7)
((9463.312755660652d0 NIL))
(interval-constraint begin-plan offload-bulk-r8)
((8021.541821038183d0 NIL))
(interval-constraint begin-plan offload-bulk-r9)
((NIL NIL))
(interval-constraint begin-plan offload-bulk-r10)
((9616.531954326058d0 NIL))
(interval-constraint begin-plan offload-bulk-r11)
((NIL NIL))
(interval-constraint begin-plan offload-bulk-r12)
((9522.950329280127d0 NIL))
(interval-constraint begin-plan offload-bulk-r13)
((NIL NIL))
(interval-constraint begin-plan offload-bulk-r14)
((15094.386768299983d0 NIL))
(interval-constraint begin-plan offload-bulk-r15)
((NIL NIL))
(interval-constraint begin-plan offload-bulk-r16)
((12268.848873064388d0 NIL))
(interval-constraint begin-plan offload-bulk-r17)
((NIL NIL))
(interval-constraint begin-plan offload-bulk-r18)
((2390.4335837734575d0 NIL))
(interval-constraint begin-plan offload-bulk-r19)
((9424.672613400595d0 NIL))
(interval-constraint begin-plan offload-bulk-r20)
((5151.635546686781d0 NIL))
(interval-constraint begin-plan offload-bulk-r21)
((5327.877300434313d0 NIL))
(interval-constraint begin-plan offload-bulk-r22)
((2198.102837419547d0 NIL))
(interval-constraint begin-plan offload-bulk-r23)
((NIL NIL))
(interval-constraint begin-plan offload-bulk-r24)
((9631.125683313552d0 NIL))
```

```
(interval- constraint begin-plan offload-oversize-r1)
((NIL NIL))
(interval-cons :aint begin-plan offload-oversize-r2)
((16600.798161788025d0 NIL))
(interval-constraint begin-plan offload-oversize-r3)
((NIL NIL))
(interval-constraint begin-plan offload-oversize-r4)
((41239.90773796092d0 NIL))
(interval-constraint begin-plan offload-oversize-r5)
((NIL NIL))
(interval-constraint begin-plan offload-oversize-r6)
((NIL NIL))
(interval-constraint begin-plan offload-oversize-r7)
((8023.312755660652d0 NIL))
(interval-constraint begin-plan offload-oversize-r8)
((9461.541821038183d0 NIL))
(interval-constraint begin-plan offload-oversize-r9)
((9367.036948896688d0 NIL))
(interval-constraint begin-plan offload-oversize-r10)
((NIL NIL))
(interval-constraint begin-plan offload-oversize-r11)
((NIL NIL))
(interval-constraint begin-plan offload-oversize-r12)
((NIL NIL))
(interval-constraint begin-plan offload-oversize-r13)
((9672.190544281659d0 NIL))
(interval-constraint begin-plan offload-oversize-r14)
((NIL NIL))
(interval-constraint begin-plan offload-oversize-r15)
((13701.452974044103d0 NIL))
(interval-constraint begin-plan offload-oversize-r16)
((12268.848873064388d0 NIL))
(interval-constraint begin-plan offload-oversize-r17)
((13708.848873064388d0 NIL))
(interval-constraint begin-plan offload-oversize-r18)
((NIL NIL))
(interval-constraint begin-plan offload-oversize-r19)
((NIL NIL))
(interval-constraint begin-plan offload-oversize-r20)
((NIL NIL))
(interval-constraint begin-plan offload-oversize-r21)
((5327.877300434313d0 NIL))
(interval-constraint begin-plan offload-oversize-r22)
((2198.102837419547d0 NIL))
(interval-constraint begin-plan offload-oversize-r23)
((3635.319543430739d0 NIL))
```

```
(interval-constraint begin-plan offload-oversize-r24)
((NIL NIL))
```

```
(interval-constraint begin-plan offload-outsize-r1)
((8262.81371196148d0 NIL))
(interval-constraint begin-plan offload-outsize-r2)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r3)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r4)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r5)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r6)
((6855.825689616498d0 NIL))
(interval-constraint begin-plan offload-outsize-r7)
((8142.573169850175d0 NIL))
(interval-constraint begin-plan offload-outsize-r8)
((8140.900620446979d0 NIL))
(interval-constraint begin-plan offload-outsize-r9)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r10)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r11)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r12)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r13)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r14)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r15)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r16)
((12392.246168042273d0 NIL))
(interval-constraint begin-plan offload-outsize-r17)
((13832.246168042273d0 NIL))
(interval-constraint begin-plan offload-outsize-r18)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r19)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r20)
((NIL NIL))
(interval-constraint begin-plan offload-outsize-r21)
```

```
((9756.884132714510 NIL))
(interval-constraint begin-plan offload-outsize-r22)
((5200.986023462708510 NIL))
(interval-constraint begin-plan offload-outsize-r23)
((3758.3573568587376d0 NIL))
(interval-constraint begin-plan offload-outsize-r24)
((9741.063160634334d0 NIL))
```

```
(interval-constraint begin-plan offload-pax-r1)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r2)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r3)
((28246.40970810148d0 NIL))
(interval-constraint begin-plan offload-pax-r4)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r5)
((29640.3386943972d0 NIL))
(interval-constraint begin-plan offload-pax-r6)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r7)
((9463.312755660652d0 NIL))
(interval-constraint begin-plan offload-pax-r8)
((9461.541821038183d0 NIL))
(interval-constraint begin-plan offload-pax-r9)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r10)
((8176.5319543260575d0 NIL))
(interval-constraint begin-plan offload-pax-r11)
((9570.885090217042d0 NIL))
(interval-constraint begin-plan offload-pax-r12)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r13)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r14)
((16534.386768299984d0 NIL))
(interval-constraint begin-plan offload-pax-r15)
((16581.45297404410 3d0 NIL))
(interval-constraint begin-plan offload-pax-r16)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r17)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r18)
((2271.6355466867810 NIL))
```

```
(interval-constraint begin-plan offload-pax-r19)
((8008.22923980178 35d0 NIL))
(interval-constraint begin-plan offload-pax-r20)
((8031.635546686781d0 NIL))
(interval-constraint begin-plan offload-pax-r21)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r22)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r23)
((NIL NIL))
(interval-constraint begin-plan offload-pax-r24)
((NIL NIL))


(interval-constraint begin-plan end-plan)
((41239.90773796092d0 NIL))
```

```
($F (LOAD-DESIGNATOR R24) (ONLOAD-STATION KSLC) (OFFLOAD-STATION EGUL)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C007)
(PRIORITY 001) (BULK-CARGO 0) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 15) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R23) (ONLOAD-STATION KMSP) (OFFLOAD-STATION EGUN)
(AVAILABLE-TIME C010) (EARLIEST-ARRIVAL-TIME C012) (LATEST-ARRIVAL-TIME C015)
(PRIORITY 001) (BULK-CARGO 200) (OVERSIZE-CARGO 150) (OUTSIZE-CARGO 0) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R22) (ONLOAD-STATION KMCF) (OFFLOAD-STATION EDAS)
(AVAILABLE-TIME C015) (EARLIEST-ARRIVAL-TIME C020) (LATEST-ARRIVAL-TIME C022)
(PRIORITY 001) (BULK-CARGO 50) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 0) (PAX 250)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R21) (ONLOAD-STATION KSKF) (OFFLOAD-STATION LETO)
(AVAILABLE-TIME C020) (EARLIEST-ARRIVAL-TIME C025) (LATEST-ARRIVAL-TIME C027)
(PRIORITY 001) (BULK-CARGO 0) (OVERSIZE-CARGO 200) (OUTSIZE-CARGO 0) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R20) (ONLOAD-STATION KMCF) (OFFLOAD-STATION LETO)
(AVAILABLE-TIME C015) (EARLIEST-ARRIVAL-TIME C020) (LATEST-ARRIVAL-TIME C023)
(PRIORITY 001) (BULK-CARGO 50) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 0) (PAX 50)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R19) (ONLOAD-STATION KSBD) (OFFLOAD-STATION EDAH)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C007)
(PRIORITY 001) (BULK-CARGO 0) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 35) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R18) (ONLOAD-STATION KCHS) (OFFLOAD-STATION EDAH)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C007)
(PRIORITY 001) (BULK-CARGO 3) (OVERSIZE-CARGO 20) (OUTSIZE-CARGO 20) (PAX 50)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R17) (ONLOAD-STATION KMCF) (OFFLOAD-STATION LETO)
```

```
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C007)
(PRIORITY 001) (BULK-CARGO 10) (OVERSIZE-CARGO 5) (OUTSIZE-CARGO 80) (PAX 50)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R16) (ONLOAD-STATION KLFI) (OFFLOAD-STATION EGUN)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C007)
(PRIORITY 1) (BULK-CARGO 0) (OVERSIZE-CARGO 250) (OUTSIZE-CARGO 0) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R15) (ONLOAD-STATION KSBD) (OFFLOAD-STATION EGUN)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C007)
(PRIORITY 1) (BULK-CARGO 100) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 0) (PAX 200)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R14) (ONLOAD-STATION KSLC) (OFFLOAD-STATION EDAR)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C007)
(PRIORITY 1) (BULK-CARGO 0) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 0) (PAX 370)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R13) (ONLOAD-STATION KTIK) (OFFLOAD-STATION EDAR)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C007)
(PRIORITY 1) (BULK-CARGO 300) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 0) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R12) (ONLOAD-STATION KTCM) (OFFLOAD-STATION EGUN)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C007)
(PRIORITY 1) (BULK-CARGO 0) (OVERSIZE-CARGO 200) (OUTSIZE-CARGO 0) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R11) (ONLOAD-STATION KJFK) (OFFLOAD-STATION EGUN)
(AVAILABLE-TIME C006) (EARLIEST-ARRIVAL-TIME C011) (LATEST-ARRIVAL-TIME C011)
(PRIORITY 1) (BULK-CARGO 100) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 0) (PAX 350)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R10) (ONLOAD-STATION KJFK) (OFFLOAD-STATION EDAF)
(AVAILABLE-TIME C007) (EARLIEST-ARRIVAL-TIME C010) (LATEST-ARRIVAL-TIME C010)
(PRIORITY 1) (BULK-CARGO 0) (OVERSIZE-CARGO 5) (OUTSIZE-CARGO 0) (PAX 350)
```

```
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R9) (ONLOAD-STATION KWRI) (OFFLOAD-STATION EDAF)
(AVAILABLE-TIME C008) (EARLIEST-ARRIVAL-TIME C009) (LATEST-ARRIVAL-TIME C009)
(PRIORITY 1) (BULK-CARGO 20) (OVERSIZE-CARGO 15) (OUTSIZE-CARGO 55) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R8) (ONLOAD-STATION KWRI) (OFFLOAD-STATION EDAF)
(AVAILABLE-TIME C006) (EARLIEST-ARRIVAL-TIME C010) (LATEST-ARRIVAL-TIME C010)
(PRIORITY 1) (BULK-CARGO 0) (OVERSIZE-CARGO 5) (OUTSIZE-CARGO 80) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R7) (ONLOAD-STATION KSTL) (OFFLOAD-STATION EDAF)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C001) (LATEST-ARRIVAL-TIME C001)
(PRIORITY 1) (BULK-CARGO 75) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 0) (PAX 370)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R6) (ONLOAD-STATION KSTL) (OFFLOAD-STATION EGUN)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C005)
(PRIORITY 1) (BULK-CARGO 100) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 0) (PAX 310)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R5) (ONLOAD-STATION KSTL) (OFFLOAD-STATION EDAF)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C004) (LATEST-ARRIVAL-TIME C004)
(PRIORITY 1) (BULK-CARGO 50) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 0) (PAX 370)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R4) (ONLOAD-STATION KSUU) (OFFLOAD-STATION EGUN)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C004) (LATEST-ARRIVAL-TIME C004)
(PRIORITY 1) (BULK-CARGO 10) (OVERSIZE-CARGO 5) (OUTSIZE-CARGO 50) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R3) (ONLOAD-STATION KDOV) (OFFLOAD-STATION EDAF)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C002) (LATEST-ARRIVAL-TIME C002)
(PRIORITY 1) (BULK-CARGO 10) (OVERSIZE-CARGO 10) (OUTSIZE-CARGO 75) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
```

```
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R2) (ONLOAD-STATION KDOV) (OFFLOAD-STATION EDAR)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C003) (LATEST-ARRIVAL-TIME C003)
(PRIORITY 1) (BULK-CARGO 0) (OVERSIZE-CARGO 20) (OUTSIZE-CARGO 30) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
($F (LOAD-DESIGNATOR R1) (ONLOAD-STATION KSUU) (OFFLOAD-STATION EDAF)
(AVAILABLE-TIME C000) (EARLIEST-ARRIVAL-TIME C005) (LATEST-ARRIVAL-TIME C005)
(PRIORITY 1) (BULK-CARGO 10) (OVERSIZE-CARGO 0) (OUTSIZE-CARGO 60) (PAX 0)
(MIN-LAUNCH-INTERVAL "0030") (MAX-LAUNCH-INTERVAL "0400") (TYPE-OFFLOAD
ENGINE-RUNNING) (ONLOAD-LOCATION-CODE ORIGIN) (OFFLOAD-LOCATION-CODE
DESTINATION) (MISSION-PREFIX NIL) (ACFT-PERMISSION-TYPE NONE)
(ACFT-CATEGORY-CODES NIL))
```

Bibliography

1. ADANS Functional Description, HQ MAC/DD-ADANS, Scott AFB, IL 62225-5001, 30 November 1988.

2. Dechter, Rina. Meiri, Itay. Pearl, Judea., Temporal Constraint Networks, *Proceedings, First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Canada, May 1989.

3. Dechter, Rina., Pearl, Judea., Tree-Clustering Schemes for Constraint-Processing, Cognitive Systems Laboratory, Computer Science Department, University of California, Los Angeles, CA 90024.

4. Doehne, T.A., Kissmeyer, K.Y., MACPLAN User's Guide, HQ MAC, Scott AFB, IL 62225

5. de Kleer, Johan., A Comparison of ATMS and CSP Techniques, *Proceedings, Eleventh International Joint Conference on Artificial Intelligence*, Detroit, Michigan, August, 1989.

6. Kohane, Isaac S., Temporal Reasoning in Medical Expert Systems, Massachussetts Institute of Technology, MIT/LCS/TR-389, May 1987.

7. Rowe, Neil C. *Artificial Intelligence Through Prolog*, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1988.

Vita

Jeffery Dean Clay was born on March 30, 1962 in Pineville, West Virginia as the youngest of three sons. He attended Pineville High School, Pineville, WV. While at Pineville High, he lettered in tennis and was active in the student council, being voted the student council president in his senior year. He graduated in 1980 as valedictorian with a 4.0/4.0 grade point average and an electronics technology degree from the Wyoming County Vocational-Technical Center.

Capt Clay entered West Virginia Institute of Technology in the fall of 1980 as an Electrical Engineering student. He enlisted in the Air Force on September 14, 1983 under the College Senior Engineering Program (CSEP) and graduated summa cum laude with a B.S. in Electrical Engineering in May 1984.

Upon graduation, Capt Clay attended Officer Training School at Lackland AFB, TX and was commissioned a second lieutenant on August 28, 1984. His Air Force career as an officer began at the Human Resources Laboratory (AFHRL) at Wright-Patterson AFB, OH. While at AFHRL, he began work on research and development to automate technical orders. He managed the Computer-based Maintenance Aids System (CMAS) and was the assistant program manager for the Integrated Maintenance Information System (IMIS).

In June 1988, Capt Clay was reassigned to the Air Force Institute of Technology (AFIT) at Wright-Patterson AFB to obtain a Master of Science degree in Computer Engineering. While at AFIT, he specialized in Artificial Intelligence.

Permanent address: P.O. Box 77
Saulsville, West Virginia
25876

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCE/ENG/89D-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology WPAFB, OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AI Technology Office 8b. OFFICE SYMBOL (If applicable) AFDC/TXI	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) WPAFB, OH 45433	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)

TEMPORAL CONSTRAINT PROPAGATION FOR AIRLIFT PLANNING ANALYSIS

12. PERSONAL AUTHOR(S)
Jeffery D. Clay, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1989, December	120

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
12	05		Temporal Constraints Constraint Propagation
12	09		Airlift Planning

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Charles R. Fisher, Lt Col, USAF	22b. TELEPHONE (Include Area Code) (513) 255-3265 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 Previous editions are obsolete SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

ABSTRACT

Developing efficient airlift plans for large operations is difficult even for experienced planners. Time is often critical and days or hours may make the difference between success and failure. Airlift plans are developed and refined through a repetitive cycle to produce usable schedules. A planner selects resources for a plan, develops a trial schedule, and analyzes the schedule for weaknesses. This process is very time-consuming and a method is needed to analyze airlift plans and provide useful feedback early in the planning process. Temporal reasoning provides a general mechanism for such analysis. Different types of temporal constraints can be inserted into a network of airlift events to provide time bounds on execution of the complete plan. For this purpose we developed a general temporal constraint reasoner and a set of mechanisms for deriving temporal information from airlift requirements and partial schedule specifications. Physical limitations of the aircraft and operating facilities as well as the availability of cargo all provide constraints on when certain events may occur. These constraints may be the time required to fly from one location to another or the time spent waiting for an aircraft to be loaded. Comparing cargo requirements with airlift capacity over time provides additional constraints. The advantage of using a temporal constraint network as the underlying representation is its ability to accommodate various sources of information about time relationships between events in a plan. By asserting temporal information about specific events in an airlift plan, the planner can assess the impact of high-level planning decisions.